Palo Alto Research Centers

The Organization of Expert Systems: A Prescriptive Tutorial

Mark Stefik
Janice Aikins
Robert Balzer
John Benoit
Lawrence Birnbaum
Frederick Hayes-Roth
Earl Sacerdoti

XEROX

The Organization of Expert Systems: A Prescriptive Tutorial

BY Mark Stefik, Janice Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti*

January 1982

VLSI-82-1

© Copyright 1982 by Mark Stefik

XEROX

PALO ALTO RESEARCH CENTERS
3333 Coyote Hill Road / Palo Alto / California 94304

ABSTRACT

Problems in reasoning are diverse. Solving them requires substantial and varied capabilities of problem solvers. This tutorial is about the design of expert systems, that is, systems for solving problems that require substantial expertise. It focuses on principles for matching the organization of an expert system to the difficulties posed by its problems.

The tutorial begins with a brief review of standard topics from artificial intelligence, such as the use of the predicate calculus and search techniques. It also discusses some newer topics such as the use of abstractions, assumptions, dependency information and meta-level cognition. These topics are essential for understanding current work on expert systems.

The main section of the tutorial is a discussion of organizational prescriptions for problem solvers. It begins with a restricted class of problems that admits a very simple organization. This organization is appropriate when the input data and knowledge are static and reliable and the solution space is small enough to search exhaustively. Few real world problems satisfy all of these requirements. The requirements are then relaxed, one at a time, yielding ten case studies of organizational prescriptions. The first cases describe techniques for dealing with unreliable data and time-varying data. Other cases illustrate techniques for creating abstract solution spaces and using multiple lines for reasoning. The prescriptions are compared for their coverage and illustrated by examples from recent expert systems.

*AUTHORS AFFILIATIONS

Mark Stefik, Xerox Palo Alto Research Center, Palo Alto, California 94304
Janice Aikins, Hewlett-Packard, Palo Alto, California 94304
Robert Balzer, USC/Information Sciences Institute, Marina del Rey, California 90291
John Benoit, The MITRE Corporation, Westgate Research Park, McLean, Virginia 22102
Lawrence Birnbaum, Computer Science Dept., Yale University, New Haven, Connecticut 06520
Frederick Hayes-Roth, Teknowledge, Palo Alto, California 90305
Earl Sacerdoti, Machine Intelligence Corp, Palo Alto, California 94303

FORWARD

Systems that exhibit "expert knowledge" have become an important product of artificial intelligence research. Knowledge is now being captured and mechanized in expert systems in many areas of expertise, including medical diagnosis, geological exploration, experiment planning, and electronic system design. With increasing commercial interest in the field, many people are seeking information about current methods of construction and application of such systems.

This timely tutorial is written against a background of rapidly accumulating experience with expert systems. Drawing on material from a widely scattered collection of research publications, the tutorial presents a unified survey of key architectural concepts. These are presented in a sequence that provides easy access for a newcomer.

In any developing technology, there are myths and misconceptions that persist while the practice is being refined. This tutorial takes a fresh and careful look at some of the "common wisdom" about expert system technology and is helpful for seeing through some of the myths of the field. For example, a common expectation about expert systems is that they should be organized into two separate components: knowledge bases and inference engines. This idea became popular after the example of the MYCIN system. The widespread interest in MYCIN was due in part to the apparent ease of adding or modifying rules in its knowledge base without changing the inference engine. The additivity of the rules was seen as a property of MYCIN's system organization and led to an expectation that expert systems should have this property. However, as more complex styles of reasoning were incorporated in expert systems, the MYCIN organization became less feasible, leading to a controversy about separability. The last section of the tutorial defuses much of this by providing examples of a more flexible approach: knowledge compilation.

We are presently expanding our knowledge engineering and expert system research activities at Xerox PARC. One key project is evolving an expert assistant for VLSI designers. This has provided us methods and an environment for substantial work on the engineering of knowledge about VLSI design methodology and design-generation heuristics. It's also led us into the development of new expert system technology for knowledge representation. As a result of this work, we've developed a keen awareness of the many opportunities for discovery and progress now open in this field.

We are very pleased to reprint this tutorial to bring expert system material to a wider audience and to encourage interactions within the larger community of researchers. I think you'll find something of the spirit, thought style, and methods of research of the field captured in the following pages. The document itself is an example of a rapid and intense collaboration by members of the AI community drawn from many different research organizations. This tutorial will help many others in the computer science community learn about, and possibly join, this exciting new work.

Lynn Conway
Palo Alto, California
29 December 1981

PREFACE

In August 1980, the National Science Foundation and the Defense Advanced Research Projects Agency co-sponsored a workshop on expert systems in San Diego. The conference organizers were Fredrick Hayes-Roth, Donald Waterman, and Douglas Lenat. The purpose of this workshop was to bring together researchers from many different institutions to write a definitive book [Hayes-Roth82] on expert systems. The participants were organized in groups corresponding to book chapters. This report will appear as a chapter in the book. A shorter version, without the introductory material in Section 2, has already appeared in *Artificial Intelligence* [Stefik82].

This paper is the product of the "architecture" group, of which Mark Stefik was the appointed chairman at the conference. The other members of the group were the other co-authors of this chapter. During the conference we struggled intensively for three days to organize our ideas. The task of writing this tutorial fell to the chairman and was carried out over the next several months. Credit, however, belongs to all of the members of the group for actively and generously pulling together to define the scope of this tutorial, sharpen distinctions, organize the ideas, and critique versions of the text.

Thanks also to Daniel Bobrow, John Seeley Brown, Johan de Kleer, Richard Fikes, Adele Goldberg, Richard Lyon, John McDermott, Allen Newell, and Christopher Tong for reading early versions of this text and providing many helpful suggestions. Thanks to Terri Doughty for help in preparing the figures and manuscript. Thanks to Lynn Conway for encouraging this work, and to the Xerox Corporation for providing the intellectual and computational environment in which it could be done.

Mark Stefik
Palo Alto, California
29 December 1981

CONTENTS

1. Introduction 1

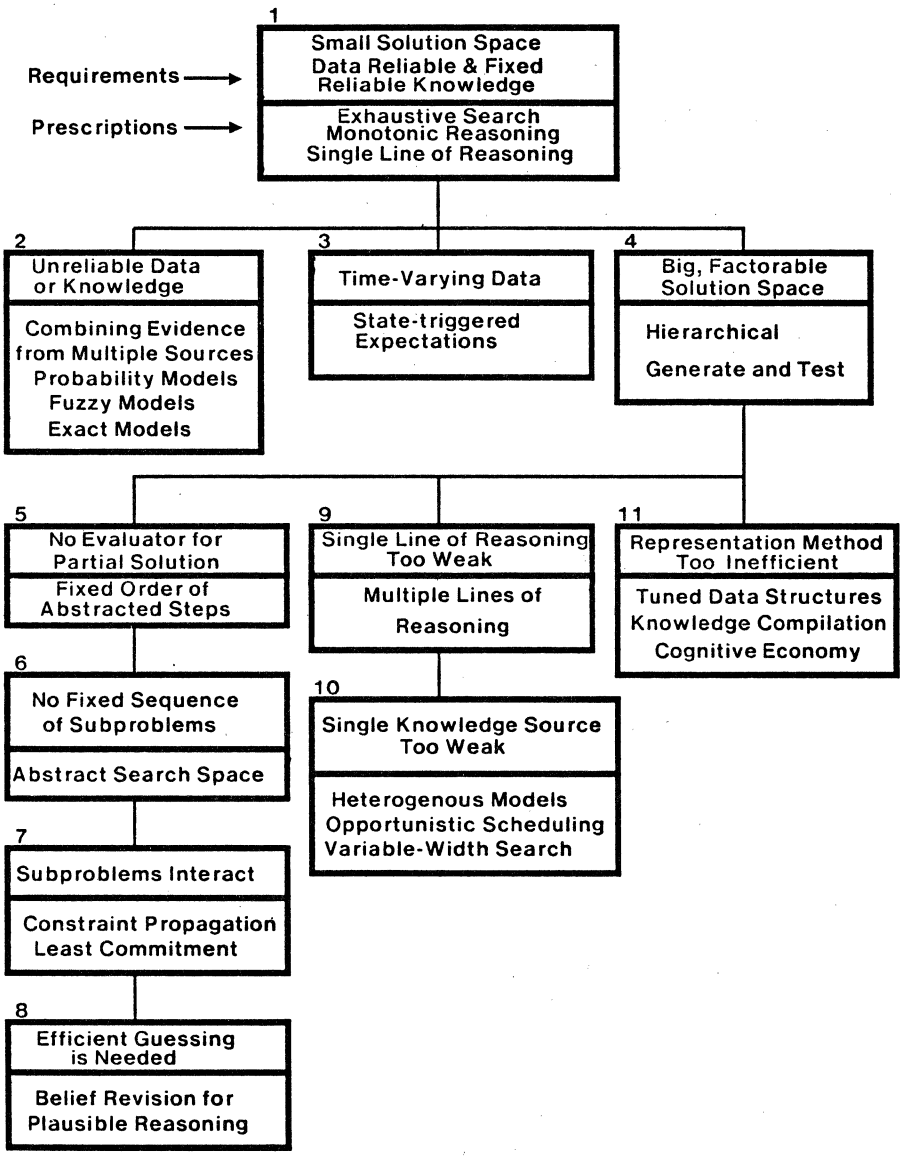
2. Vocabulary and Basic Concepts 3
   - Symbols 3
     - Predicate Calculus 4
     - Inference 6
   - Search 8
     - State-Space Search 8
     - Blind Search 10
     - Heuristic Search 10
     - Abstracting the Solution Space 11
     - Generate and Test 12
   - Symbolic Reasoning 13
     - Assumptions and Commitments 14
     - Dependencies and Justifications 15
     - Constraints and Goals 18
     - Factoring into meta-Problems 20
   - Summary of Basic Concepts 22

3. A Characterization of Expert Tasks 23

4. Knowledge Engineering Prescriptions 27
   - Small Search Space with Reliable Knowledge and Data 28
   - Unreliable Data or Knowledge 29
   - Time-varying Data 32
   - Large but Factorable Search Space 33
   - No Evaluator for Partial Solutions 36
   - No Fixed Partitioning of Subproblems 38
   - Interacting Subproblems 40
   - Guessing is Needed 43
   - Single Line of Reasoning is Too Weak 47
   - Single Source of Knowledge is Too Weak 48
   - General Representation Methods are Too Inefficient 51

5. Summary of Knowledge Engineering Cases 55

References 57



Expert System Prescriptions
(See Section 4)

Case 1 begins with a restricted class of problems that admits a very simple organization. In the other cases, these assumptions are relaxed one at a time.

1. INTRODUCTION

Twenty years ago, Newell [Newell62] surveyed several organizational alternatives for problem-solving programs. Many techniques have been developed in artificial intelligence (henceforth AI) research since then and several examples of *expert systems* have been built. Expert systems are problem-solving programs that solve substantial problems generally conceded as being difficult and requiring expertise. They are called *knowledge based* because their performance depends critically on the use of facts and heuristics used by experts. Expert systems have been used as a vehicle for AI research under the rationale that they provide a forcing function for research issues in problem-solving and a reality test for their workability.

Some textbooks discuss principles of AI (e.g., Nilsson [Nilsson80]) and give examples of advanced programming techniques (e.g., Charniak *et al.* [Charniak80]). However, there is no guidebook for an aspiring expert systems architect to guide him through the issues and choices that he faces in designing a system. Furthermore, an unguided sampling of expert systems from the literature can be quite confusing. Examples are scattered in diverse journals, conference proceedings, and technical reports. Systems with seemingly similar tasks sometimes have radically different organizations, and seemingly different tasks are sometimes performed with only minor variations on a single organization. The variations often reflect the immaturity of the field in which most of the systems are experimental. From the diversity of experiments one would like to extract alternatives and principles to guide a designer. This is the subject of this tutorial.

This presentation is intended for readers with a computer science background but not necessarily an AI background. It is organized as follows: Section 2 is an introduction to the concepts and vocabulary of AI. Readers who have examined the AI literature may encounter some familiar examples. The last part of this section contains some material that is not usually covered in an introduction to AI, but which seems important for understanding current research in expert systems. Section 3 considers a variety of *expert* tasks and provides a checklist of requirements and key problems. The purpose of this section is to extract a broad set of architecturally relevant issues. Section 4 presents the substance of the architectural ideas. It provides a pedagogical tour of prescriptions for the organization of expert systems (see Frontispiece). Examples are drawn from expert systems developed over the last ten years. Section 4 compares and evaluates approaches and attempts to organize the ideas into a coherent theory.

2. VOCABULARY AND BASIC CONCEPTS

In their 1975 Turing Award lecture, Newell and Simon emphasized two basic concepts: symbols and search. They were concerned with "physical symbol systems", which manipulate collections of symbolic structures and perform problem-solving tasks using heuristic search. Their central hypothesis was that physical symbol systems have all of the necessary and sufficient means for intelligent action. Since expert systems are intended to embody knowledge and intelligence from experts, the concepts of symbols and search are central. Our emphasis, however, will be on the pragmatics of building expert systems and not on the theoretical requirements for intelligence.

This section starts with a discussion of symbol structures that can be used to represent facts. It reviews terminology from predicate calculus. Several approaches to search are compared for their effectiveness in inferring new facts from old ones. Some characteristics of common sense reasoning that impact the computational organization of a problem-solver are considered. This section presents several concepts that are not covered in introductory AI textbooks, but which are important in expert systems.

2.1 Symbols

In their discussion of physical symbol systems, Newell and Simon [Newell75] define a symbol as a physical pattern that can occur as a component of a symbol structure. A symbol structure is composed of a number of symbols related in some physical way, such as being next to each other. For our purposes, it will usually be sufficient to think of symbols as strings of characters and symbol structures as a type of data structures called *list structures* containing symbols. The following are examples of symbols:

Apple
Transistor-13
Running
Five
3.14159

and the following are examples of symbol structures:

(On Block1 Block2)
(Plus 5 X)
(Same-as (Father-of Pete) (Father-of (Brother-of Pete)))

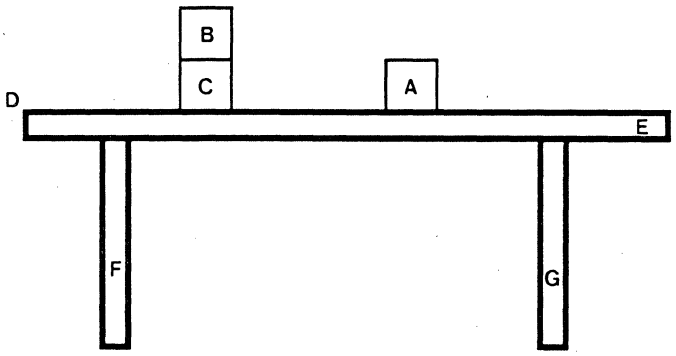
One of the early contributions of artificial intelligence research to computer science was the invention of list processing languages for symbolic computation. These languages provided primitives for manipulating lists and facilities for managing their storage (e.g., Charniak *et al.* [Charniak80]). Our discussion of symbols and symbol structures will emphasize how they can be used to represent knowledge.

2.1.1 *Predicate Calculus*

The predicate calculus is a widely studied formal language of symbol structures which can be used for representation in a computer. This section introduces some concepts of the predicate calculus that are relevant to symbolic computing. It starts with simple examples of the use of flexible symbol structures for representing facts and leads to an illustration of why inference methods must be *knowledge-intensive* to be effective.

Although the issues for reasoning in expert systems go beyond the issues of classical logic, a knowledge of the predicate calculus is an essential foundation for understanding issues of representation and inference [Kleene67 and Suppes57].

Figure 1 shows a picture of a table with some blocks on it. Some symbol structures are shown that represent the information in the picture. The symbol structures are written in a syntactic variation (prefix format) of predicate calculus. They are made up of terms and predicate symbols. Terms are used for the names of things and predicates represent relations between things. In this example, A, B, C, D, E, F, G, BLOCK, TABLE, TABLE-TOP, and TABLE-LEG are terms and IS-A, PART-OF, and ON are predicate names. In books on logic, simple predicates like those in Figure 1 are called propositions or atomic formulas.



(IS-A A BLOCK)
(IS-A B BLOCK)
(IS-A C BLOCK)
(IS-A D TABLE)
(IS-A F TABLE-LEG)
(IS-A G TABLE-LEG)

(PART-OF E D)
(PART-OF F D)
(PART-OF G D)
(ON A D)
(ON C D)
(ON B C)

Figure 1. A table with blocks on it and a predicate calculus representation for the information

In addition to symbols for constants and predicates, predicate calculus allows for functions and logical connectives. Functions denote mappings between entities. For example, if **TOP-OF** is a function symbol, then

(TOP-OF D)

denotes the top of table D. (Our syntax differs from the usual predicate calculus only in the placement of the predicate symbol. The prefix syntax was chosen to resemble list structures

convenient in symbol processing languages such as LISP.) Connectives are used to combine formulas. These include "$\wedge$" (and), "$\vee$" (or), and "$\rightarrow$" (implies). For example, the formula "Either Block A is on table D or it is on Block B" might be represented as:

ON (A D) $\vee$ ON (A B)

in functional notation or as

(OR (ON A D) (ON A B))

in our notation. The symbol "$\sim$" (not) is sometimes called a connective, although it is used to negate a formula and not to connect two formulas.

Predicate calculus can also express sentences involving quantifiers like "All blocks are small". In traditional notation, this would be expressed as

$(\forall x)$ [BLOCK(x) $\rightarrow$ SMALL(x)] .

A list structure notation for this could be

(ALL (x) (IF (IS-A x BLOCK) (SMALL x))) or

(ALL (x) ((IS-A x BLOCK) $\rightarrow$ (SMALL x))).

These formulas use "$\forall$" (all) the universal quantifier to indicate that the formula is true for all assignments of the variable x to entities in the domain of discourse. We say that the formula is quantified over x. The second important quantifier in predicate calculus is the existential quantifier "$\exists$" (exists). This quantifier is used to indicate that the formula is true for some assignment of the variable to entities in a domain. A calculus is said to be first order if it allows quantification over terms, but not over predicate or function symbols.

Formulas that have been built up out of terms and atomic formulas by using connectives and quantifiers are called *wffs* (or well-formed formulas). The predicate calculus provides a number of well-defined rules for combining formulas.

In the predicate calculus, a wff can be given an interpretation by assigning a correspondence between the elements of the language and the entities and relations of the domain of discourse. To each constant symbol we must assign a corresponding entity in the domain, and to each predicate we assign a relation in the domain. When a computer data base is composed of symbolic representations, the designer of the data base selects a vocabulary of predicates and terms that he will use and decides what they will mean. Use of the data base follows an interpretation of the formulas as assertions in the domain of discourse. For example, the formulas in Figure 1 are meant to be interpreted as facts about the blocks and table in the illustration. The formula **(PART-OF E D)** is intended to represent the fact that **E** (the table top) is part of **D** (the table).

The strength of predicate calculus is that the language has well understood interpretations to express many involved sentences. De Morgan's laws provide an example of manipulations that preserve standard interpretations. For example,

(NOT (AND A B))

is equivalent to

(OR (NOT A) (NOT B))

in the predicate calculus. Another example is the use of parentheses for indicating scoping in quantification. The two formulas

(ALL (x) (SOME (y) (Loves x y)))

(SOME (x) (ALL (y) (Loves x y)))

mean respectively "Everyone has someone that he loves" and "There is someone who loves everybody".

2.1.2 *Inference*

In order for a system to reason, it must be able to infer new facts from what it has been told already. This involves dynamically creating new symbol structures from old ones. In the predicate calculus, new wffs are produced by applying *rules of inference* to sets of wffs. In the situation in Figure 1 for example, we might have the following rules about the predicate **ABOVE**:

(ALL (x) (ALL (y) ((ON x y) → (ABOVE x y))))
"If x is on y, then x is above y."

(ALL (x) (ALL (y) (ALL (z) (AND (ABOVE x y) (ABOVE y z)) → (ABOVE x z))))
"If x is above y and y is above z, then x is above z."

Given the wffs in Figure 1, we can use the first rule to infer that the blocks are *above* everything that they are *on*, that is:

(ABOVE A D)
(ABOVE C D)
(ABOVE B C)

The second rule allows us to utilize the transitivity of the **ABOVE** relation to infer

(ABOVE B D) .

This account is logically incomplete. The only rules that we described were *domain rules*, that is, wffs from the domain of discourse in Figure 1. We neglected to mention the logical rules of inference, that is, the rules about logic that take given wffs to yield derived wffs. In carrying out

the above reasoning, we implicitly used two rules of inference known as *modus ponens* and *universal specialization.* In standard notation, these are written:

modus ponens: $A, A \rightarrow B \models B$

Universal specialization: $A, (\forall x)\ W(x) \models W(A)$.

where the symbol "$\models$" is used to mean "yields". *Modus ponens* states that from the wffs A and $A \rightarrow B$ we can infer the wff B. Universal specification is similar except that it involves a universal quantifier. *Universal specialization* produces the wff $W(A)$ from the wff $(\forall x)\ W(x)$ where A is any constant symbol. An often cited example of universal specialization is the inference that "Socrates is mortal" produced from the wffs "Socrates is a man" and "All men are mortal".

The direct application of this to expert systems is to represent "what is believed so far" by wffs stored in computer memory and to use inference rules to derive new facts and beliefs. This is a good idea, but it falls short as a means of representing knowledge in expert systems. One of the difficulties is that it is not enough to simply have the "facts at hand", one must know how to use them. Consider, for example, the inference rule *or-introduction*:

$$A \models A \vee B.$$

Or-introduction captures the idea that we can infer "A or B" by either proving A or by proving B. Given constants D, E, and F, we can use this rule to infer

$D \vee E$
$D \vee F$
$E \vee F$

as well as such wonders as

$D \vee D$
$D \vee E \vee E$
$D \vee E \vee D \vee E$
$D \vee E \vee E \vee E \vee E \vee E \vee E \vee E$

and so on without limit. This example shows that the unguided application of inference rules can be *combinatorially explosive.* The inferences are perfectly correct. They are just not particularly interesting. (In the next section we will consider combinatorial explosions that arise in large search problems.)

Much work has been directed toward controlling combinatorial explosions. For example, some mechanical theorem proving techniques avoid "nonsense" applications of *or-introduction.* Methods that use many rules of inference need to incorporate knowledge to control their use. An alternative approach is to use a single rule of inference such as resolution. (The interested reader is referred to Nilsson [Nilsson80] for examples of resolution and resolution strategies, as well as a bibliography.) Resolution has the important property of completeness -- meaning that any wff which follows from a set of wffs can *eventually* be derived.

Unfortunately, the effort that it takes to derive a theorem with resolution is often quite impractical for even simple problems. Human problem-solvers can often solve problems using direct methods. The reason is that the knowledge about problem-solving is much more encompassing than the knowledge about the application of inference rules. In other words, the speed and directness of a human expert has little to do with his expertise about the semantics of **AND**, **OR**, and **NOT**, and more to do with his expertise about the domain and about solving problems. It is possible to incorporate problem-solving expertise in theorem proving systems. The point is that problem-solvers need such knowledge to be effective. In the next section we will approach this issue by considering the concept of search.

2.2 Search

Many problem-solving systems in artificial intelligence are based on the formulation of problem solving as search [Gardner79 and Nilsson80]. In this formulation, a description of a desired solution is called a goal and the set of possible steps leading from initial conditions to a goal is viewed as a search space. Problem-solving is carried out by searching through the space of possible solutions for ones that satisfy a goal. These concepts will be illustrated in the example that follows.

This section begins with exhaustive search and leads to more sophisticated approaches that use heuristics and abstractions to improve efficiency. These examples are important for understanding the problem-solving architectures later in the paper. Some of the examples have been drawn from textbooks in artificial intelligence.

2.2.1 *An Example of a State-space Search Problem*

The simplest search formulation of problem-solving is state-space search. This representation of problem-solving uses *states* and *operators*. States are data structures that represent snapshots of the problem at each stage of the solution. Operators change one state into another. The idea is to find a sequence of operators that can be applied to a starting state until a goal state is reached.

A simple example of a state-space search, which has been used in many introductions to AI, is the 8-puzzle. An 8-puzzle is a tray containing 8 square tiles numbered from 1 to 8. The space for a 9th tile is vacant, leaving an empty square in the tray as shown in Figure 2. The goal is to put the tray of tiles into a specified configuration by sliding tiles.

A tile may be moved by sliding it vertically or horizontally into the empty square. This movement can be formalized concisely in terms of operators by viewing the empty square as an object that can be moved in any of the four directions. This provides us with four operators -- *up*, *down*, *left*, and *right* - which can be applied to a state of the tray to yield another state. Figure 2 shows the effects of applying these operators to a starting state of the 8-puzzle. Under some circumstances, particular operators may be inapplicable. In the 8-puzzle, operators are inapplicable when the empty square is on the edge of the figure and the proposed operator would move it off the tray. A state-space is the directed graph whose nodes are states and whose arcs are the operators that lead from one state to another. There are various ways to carry out the search for a solution. Blind search techniques systematically explore branches of the graph one step at a time.

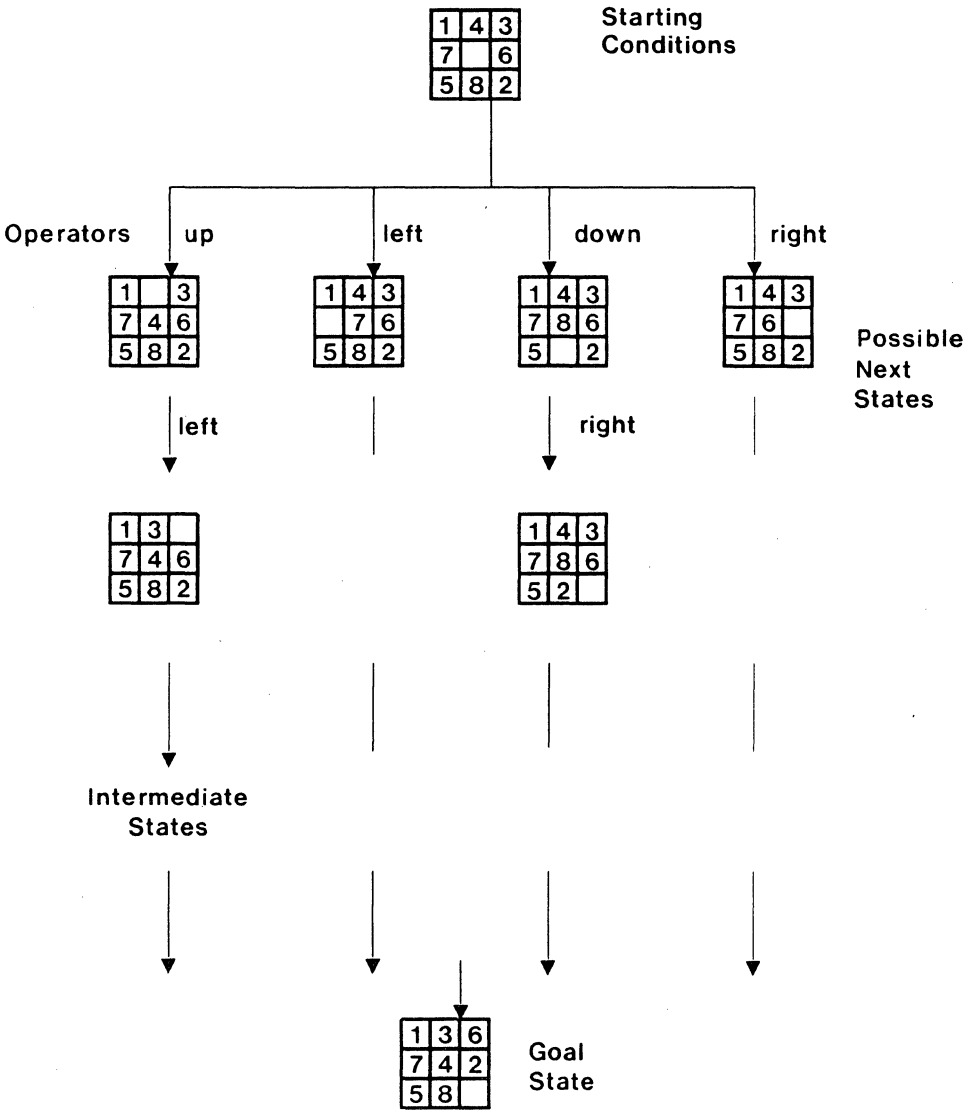


Figure 2. State-space representation of the 8-puzzle. Each board configuration is a "state" and operators are applied to move between states in the space to a goal.

2.2.2 *Blind Search*

Blind search methods can be top-down, bottom-up, or bi-directional. Top-down search means searching from goals. In the 8-puzzle described above, this would involve searching backwards from the goal state to the initial conditions by applying the "inverse" operators. Top-down search is also called searching backwards. Bottom-up search refers to search starting from given conditions. Bottom-up search is also called searching forwards. Bi-directional search involves searching from both ends of a space and (hopefully) meeting someplace in the middle.

Given an orientation for a search, there are several different systematic orders in which the nodes of the solution space may be considered. Depth-first search is a search process which considers successive nodes in the space before considering alternatives at the same level. In the 8-puzzle case, a forwards depth-first search would expand the graph of the state-space starting with one of the states in the first row, and then considering a successor to that state, and then a successor of the successor, and so on. A breadth-first search would expand the graph differently. First it would consider all four initial operators, and then it would consider all of the successors of those operators, and then all of the successors of the successors, and so on. In summary, depth-first search dives deeply into the search tree and breadth-first search descends uniformly across all possibilities. In a complete search, depth-first and breadth-first approaches will examine the same number of nodes. However, breadth-first search places a substantially greater demand on memory usage because it carries so many paths in parallel.

2.2.3 *Heuristic Search*

There are always practical limits on the amount of time and storage available to spend on search problems. Although, in principle, blind search methods can eventually find a solution to search problems, they are not practical for large problems because they visit too many nodes before a solution is found. This illustrates the phenomenon of combinatorial explosions that arise when problem solvers lack sufficient knowledge to guide an inferential process.

For many applications it is possible to find domain-specific information to guide the search process and to reduce the amount of computation. This is called *heuristic* information and search procedures which use it are called heuristic search methods. Heuristic search is appropriate if we are *satisficing*, that is, if we can stop after finding one *good* solution.

One form of heuristic search is to direct the search in a best-first order. To determine which branch of the search tree to expand, a domain-dependent *evaluation function* is used to estimate the closeness of the path to the goal. For example, Nilsson [Nilsson80] suggests the following evaluation function for the 8-puzzle:

$$\mathbf{f(n) = d(n) + W(n)}$$

where **d(n)** is the depth of node **n** in the tree and **W(n)** counts the number of misplaced tiles. An evaluation function is supposed to give an estimate of the computational effort of pursuing a path. Nodes are tried in increasing order of their **f** values. In Fig. 2, the **f** values of the branches in the first row (left to right) are 5, 7, 7, and 6 so the first move in the 8-puzzle state-space would be *up*.

Much attention has been given to formally characterizing evaluation functions, especially for games. These functions are said to be well-behaved if they reliably indicate an optimal path to a goal. The monotone restriction, for example, requires that an evaluation function must infallibly decrease as a goal is approached. Various theorems have been proved about search methods utilizing well-behaved evaluation functions [Nilsson80].

In many real problems, well-behaved evaluation functions are elusive. Sometimes a strategic retreat is necessary; that is, one must seem to move away from a goal (overriding some evaluation function) in order to achieve it. For example, to enter a room it is worth detouring to an unlocked door even if a locked door is closer but there is no key. In organic synthesis problems, it is sometimes helpful to add elaborate extra chemical structures to a molecule in order to provide a rigid framework for a later step in a series of reactions. If an evaluation function measures only chemical similarity, such reactions appear to be a retreat away from the goal.

This illustrates a basic fact of life about evaluation functions. To be useful, evaluation functions must characterize the solution space adequately and this generally requires a substantial amount of knowledge. It is misleading to consider only simple arithmetic functions and weighted coefficients as the basic style for representing this knowledge. In expert systems, substantial amounts of symbolic computing may be appropriate for guiding a search.

2.2.4 *Abstracting the Solution Space*

One of the important skills in problem solving is the ability to focus on the most important considerations in a problem. Search methods get bogged down when they "can't see the forest for the trees". The following example illustrates the idea.

Consider the problem of driving from the San Diego Hilton to the New York Hilton. We will assume that we want to take the fastest route, and that other considerations (such as scenery or weather) are unimportant. Our trip planning could be formalized as a search process which starts at the San Diego Hilton, and fans out through the streets of San Diego enumerating non-cyclic paths until we have found the fastest route to the New York Hilton. There are hundreds of roads in San Diego and hundreds of towns (with many roads) between San Diego and New York. We could apply some heuristics about heading to the north and east and not doubling back too much, but we want to be careful not to miss the fastest route, which is not likely to be along a straight line. Using these ideas and the fastest computer available we would never manage to find a route to New York.

Computers need not be so simple-mindedly methodical. In planning the trip without a computer, we would not busy ourselves with the street maps for all of the towns in the continental United States. We would start with a reduced scale map, and would consider only the main highways that cross the country. We would use the street maps only for incidentals of lodging and eating along the way. The use of a reduced scale map is essential to an efficient solution to the problem. It is an *abstraction* of a detailed continental map that leaves out the details.

This illustrates a general idea for reducing search in many kinds of problem-solving. By searching an abstracted representation of a space we can reduce the combinatorics. The search of

the abstract space is quicker because it is smaller; single steps in the abstract space correspond to big steps in the ground search space. For example, we may first decide to take route 40 across the country. This breaks the original problem down into smaller problems. The smaller problems can be tackled using intermediate *levels of abstraction.* In the driving example using a state map, we can plan to use highway 15 to meet route 40 at Los Angeles and using a city map we can plan a route out of San Diego. This amounts to *hierarchical* problem-solving through various levels of abstraction.

The importance of hierarchical methods for taming large search problems has been recognized for many years. In 1961, Minsky [Minsky61] argued that by introducing planning islands, these methods reduce search by what he called a "fractional exponent":

> "In a graph with 10 branches descending from each node, a 20-step search might involve 10^{20} trials, which is out of the question, while the insertion of just four *lemmas* or *sequential subgoals* might reduce the search to only 5 x 10^4 trials. which is within reason for machine exploration. Thus it will be worth a relatively enormous effort to find such "islands" in the solution of complex problems. Note that even if one encountered, say 10^6 failures of such procedures before success, one would still have gained a factor of perhaps 10^{10} in over-all trial reduction! *Thus practically any ability at all to "plan" or "analyze" a problem will be profitable, if the problem is difficult.* It is safe to say that all simple, unitary, notions of how to build an intelligent machine will fail, rather sharply, for some modest level of problem difficulty. Only schemes which actively pursue an analysis toward obtaining a set of *sequential goals* can be expected to extend smoothly into increasingly complex problem domains." [pp. 441-442]

Hierarchical methods have been used in several expert systems. As in the road map example, it is not necessary to introduce fully described intermediate "islands" in order to achieve hierarchical reasoning; one can use abstract descriptions.

2.2.5 *Generate and Test*

State-space search is sometimes formulated as generate and test. This formulation divides the search process into two parts: a generator of possible solutions and a tester which prunes solutions that fail to meet some constraints. A generator is said to be complete if it is capable of producing every possible solution; it is said to be non-redundant if in the course of generation it will produce each solution exactly once. In most applications, the generate and test processes overlap in time.

An important issue is the distribution of knowledge between the generator and the tester. Putting as much knowledge as possible in the generator often leads to a more efficient search. An important form of generate and test that does this is *hierarchical* generate and test. This formulation permits pruning of candidate solutions that are only partially specified (e.g, only the first half may need to be specified). When a partial description is pruned, an entire class of

solutions corresponding to the description is eliminated from the generation process (See Figure 3). Hierarchical generate and test applies powerful pruning rules early in the generation process.

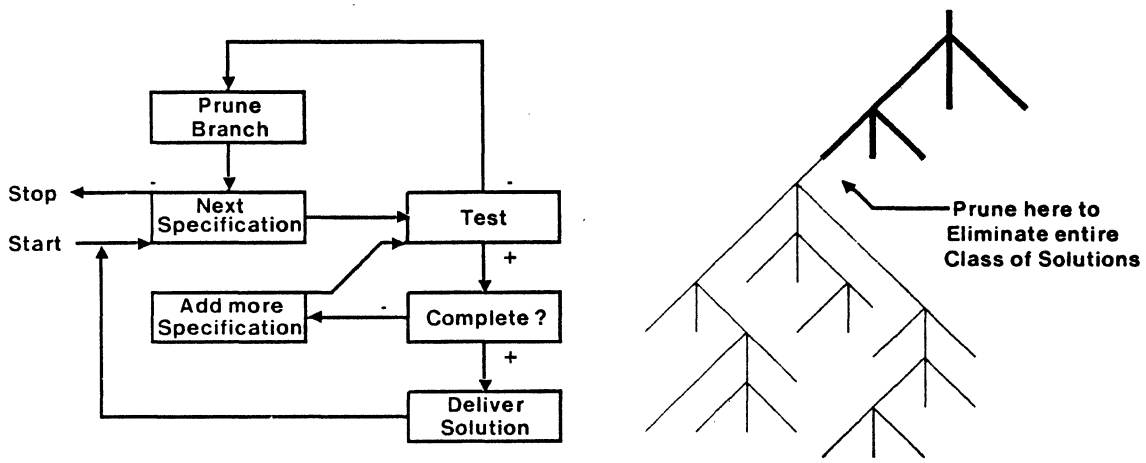


Figure 3. Hierarchical generate and test eliminates whole classes of solutions by early pruning of partial descriptions.

In summary, a search process navigating in a large space can exploit:

(1) The ability to follow plausible paths (e.g., using an evaluation function to guide an incomplete search)
(2) The ability to search abstract spaces and refine the solutions to the detailed space (i.e., hierarchical planning), or
(3) The ability to prune effectively (i.e., reasoning by class elimination in hierarchical generate and test).

The first and second methods are most applicable for problems where the goal is to satisfice. The third method is appropriate for optimizing, because it admits a complete search if the pruning rules are sufficiently powerful.

2.3 Symbolic Reasoning

In the introduction, we said that an expert systems architect must understand representation and search. These topics are traditional in sources on AI. In this section we suggest that he must also understand the organization of systems for symbolic computation. We will begin by observing that expert reasoning is often non-monotonic, that is, it requires the making and retraction of assumptions. This will lead us to consider how to account for justifications in problem solving, and how a program can treat beliefs as tentative. Then we will extend these ideas to cover goals and consider how a program can dynamically understand and direct its problem-solving activities. We will observe that such reasoning requires substantial problem-solving knowledge. One way to organize this knowledge is to view problem-solvers as working on several related meta-levels.

2.3.1 *Assumptions and Commitments*

Experts, like other people, face the need to act in spite of the lack of time, facts and knowledge. When experts solve problems, they use methods different from those of formal mathematical reasoning. In mathematical reasoning, every conclusion must follow from previous information. In contrast, common sense reasoning needs to draw conclusions from partial information. People tentatively accept plausible conclusions for which they have no "proof".

The need for common sense reasoning can be illustrated by the previous trip planning example. Suppose we want to drive our car to a restaurant. Many things are required for the car to operate correctly. For example:

> **To be ready for use, a car must:**
>
> **(1) have gasoline, and**
> **(2) have a charge in its battery, and**
> **(3) have air in its tires, and**
> **(4) have no leaks in its tires, and**
> **(5) etc.**

If we watched a person who was about to drive to the restaurant, we would not be surprised to see him glance at the fuel gauge. We would explain this fact by saying that he knew that the car needed gasoline. If he opened the hood to check the battery (or flashed the lights on and off), we would consider his behavior unusual because batteries are rarely broken. If he walked around the car to check the tires, we might be surprised. If he dismounted the tires to check for leaks, we would probably ask some questions. After all, if the tires are full of air, they probably don't have any leaks.

In each of these examples, common sense leads us to make certain assumptions. One way to think about assumptions is in terms of "default values". For example:

> **Assume that batteries are charged.**

Sometimes we know that assumptions pertain only to certain conditions. Such assumptions can be contingent on certain conditions. For example:

> **If the battery is not brand new and the lights were not left on and the generator has been working, assume that the battery is charged.**

Both cases depend on a style of reasoning for making assumptions *in the absense of contrary information.* Another rationale for making assumptions is based on a view of reasoning as a resource-limited process. An example of this would be:

> **If you can't prove otherwise (without taking the wheels off the car), assume that the tires have no leaks.**

This heuristic assumes that a fact is true *unless the contrary can be proved using limited resources.*

In these cases, assumptions and further inferences based on them must be regarded as tentative -- subject to revision given new information. This aspect of reasoning is called *non-monotonic*. Formal mathematical logic is monotonic in the sense that facts are additive; no belief need ever be revised [Winograd80].

Assumptions are also used in reasoning by means of *reductio ad absurdum*, as in the sequence:

Assume A
Prove B
Prove ~B
Retract A
Conclude ~A

Non-monotonic reasoning is also important in design and planning problems. In these problems, the space of possible solutions is sometimes very large and it is not usually possible to anticipate the consequences of a preliminary design choice. A designer knows generally what he wants, but not how to create it. In design assumption takes the form of a tentative decision. For example, the following might be the dialogue of a designer thinking through his options:

(1) Suppose we use serial transmission on the data bus.
(2) Then the bus will fit through the wiring channel.
(3) And the maximum data rate will be 6 MHz.
(4) But that's too slow.
(5) So we'd better use a parallel scheme for the bus.
...

In (1), he proposes a serial implementation. In (2) and (3) he checks to see whether some design constraints are satisfied. Since the bus would be too slow, he decides to try a parallel implementation instead. In this example, the new information was obtained by exploring the consequences of the design assumption.

In summary, common sense reasoning requires that a system be able to revise its beliefs in light of new knowledge received or derived. This suggests a view of reasoning in which beliefs are transient and subject to change.

2.3.2 *Dependencies and Justifications*

To *revise* its beliefs in response to new knowledge, a problem solver must reason about dependencies among its current set of beliefs. New beliefs can be the consequences of new information received or derived. This section considers the recording and use of dependency information for belief revision. It begins by showing how the techniques are an extension of chronological backtracking in search.

Chronological backtracking is an approach for implementing search. In this approach, search failure results in backtracking, that is, first retracting all of the actions and inferences since the most

recent choice point and then continuing with the next alternative for that choice. Because of the *last-in-first-out* order of the backtracking, the memory for storing the set of active beliefs can be implemented by a push-down stack. Chronological backtracking is often needlessly inefficient because failures independent of the search path are forgotten when the path is abandoned. These failures will be subsequently rediscovered on other search paths. The following story about a hypothetical robot illustrates the difficulty:

> **Robie wants to pick up the block.**
> **He reaches to pick it up with his right hand.**
> **But the block is hot and Robie burns his hand!**
> **Robie drops the block and backtracks (chronologically).**
> **Robie reaches to pick it up with his left hand ...**

Chronological backtracking throws away too much information. A much better idea is to trace errors and inconsistencies back to the inferential steps that created them. This requires recording the inferential steps. Such records are called dependency records. A search method that analyzes dependencies and decides what to invalidate is called *non-chronological* or *dependency-directed.*

The fundamental objects in dependency records are beliefs, inference rules, and *justifications.* Justifications represent inferences. The simplest sort of justification is a record of immediate antecedents. For example, suppose our reasoning system has the statements:

> **Belief-1:** **My car has no gasoline.**
> **Belief-2:** **If a car has no gasoline,**
> **then it will not start.**
> **Rule-3:** ***modus ponens***

From these data it might infer the new statement:

> **Belief-4:** **My car will not start.**

The justification for Node-4 identifies the logical support for this inference. It could be represented as a data base record like the following:

> **Justification-1**
> **SupportFor:** **Belief-4**
> **InferenceRule:** **Rule-3**
> **Premises:** **(Belief-1 Belief-2)**

Justifications can help maintain an active set of beliefs called a theory. This process is called belief revision [Doyle80] or "truth maintenance". All beliefs in the current theory must have valid justifications and the theory is updated whenever justifications are added or modified. A justification is valid if each of the beliefs it mentions is currently active. Some beliefs (called premises) are always active.

Belief revision can be used for reasoning with assumptions and defaults. The following scenario drawn from the rules in the previous section about starting a car shows how a belief revision system could act:

(1) Robie has the goal to drive to the restaurant.
(2) He forms a subgoal to start the car.
(3) Robie checks the fuel gauge.
(4) He turns on the ignition.
(5) The engine turns over vigorously, but won't start.
(6) Robie tries many things, but he can't figure out what is wrong.
(7) Finally, he taps the fuel gauge with his finger,
and the needle swings to empty.

This scenario illustrates several points about non-monotonic reasoning. First, there are many things that Robie could check. For example, the following statements are *not* in the story:

(3.1) Robie checks the tires for air.
(3.2) Robie checks the battery for charge by turning on the lights.

Robie does not check these things because he *assumes* that they are satisfactory. There are many possible reasons for making these assumptions. For example, Robie may know:

Unless you have evidence for flat tires, assume that they have air.

He may omit the test of the battery charge either because he assumes that the battery is charged, or because he knows that turning on the ignition will effectively test this anyway. When the car doesn't start, Robie needs to reconsider his assumptions. For example, he could try the step:

(5.1) Turn on the lights to check the battery.

Alternatively, Robie might use the rule:

If the starter turns the engine vigorously,
then the battery is charged.

This rule saves Robie the effort of testing the assumption (that the battery has a charge) and provides a new justification for that belief. In the last steps in the scenario, Robie has eliminated many possible causes for the problem. Before he can identify the real problem -- lack of gasoline -- he must suspect that the needle in his fuel gauge has become stuck. This scenario shows that Robie's beliefs are working hypotheses that are subject to change pending further information. This style of reasoning amounts to reasoning about justifications.

A critical issue in this style of reasoning is well-founded support and there are some pitfalls for the unwary involving cyclical support structures. An important question is "what mechanisms should be used to resolve ambiguities when there are several possible revisions?" It is clear that this choice needs to be controlled, but the details for making the decision remain to be worked out. In the examples above, we used knowledge *about justifications* to reason about choices. Doyle [Doyle80a] has proposed a style of dialectical argumentation where the primary step is to argue

about the kinds of support for beliefs. In such systems the complexity of knowledge about belief revision would itself require a substantial knowledge base. Every approach depends critically on the kinds of dependency records that are created and saved. This work is at the frontier of current AI research.

These examples show that the technique of storing dependency information can be important in a variety of applications. This is not to suggest that the techniques for revising beliefs are well understood. Basic algorithms are needed for making changes to the set of beliefs (or their justifications) when their support is changed. Doyle [Doyle79] and McAllester [McAllester80] present alternative approaches, but much more work needs to be done.

2.3.3 *Constraints and Goals*

Programming languages are usually organized around unidirectional computations. Programs produce desired outputs using predetermined operations on their inputs. In contrast, constraint systems are non-directional. A constraint such as:

(EQUALS Voltage (TIMES Current Resistance))

says no more about how to compute voltage than how to compute current. Rather, it defines a relationship between referents of symbols. Given the knowledge of how to solve linear equations, this constraint enables us to compute a numerical value for any of the arguments given values for the other two.

Constraints need not be numerical. For example, the following constraint describes a relationship between variables bearing on material selection in a storage battery:

(Resists Wall-Material Battery-Fluid)

If the variable **Battery-Fluid** is bound to a strong acid, this constraint places restricts the choice of values for the variable **Wall-Material**. If the wall material were already determined, the constraint could be used to limit the choices of a battery fluid. As in the electrical constraint above, some means of finding solutions that satisfy the constraint must be provided. For different kinds of problems, the solution process may range from a heuristic search to matrix methods for the solution of mathematical equations.

Constraints can be viewed as partial descriptions of entities. For example, in a circuit design system we could start with a partial description of some circuit element and then collect constraints on its implementation that are implied by the interfaces with other circuit elements. In this way, the constraints effectively augment the partial description of the circuit element.

Constraints can be used to represent goals in hierarchical search methods. For example, Figure 4 shows two representations of an "inverter" in an integrated circuit design system. The upper figure shows the symbol for an inverter at the "switch level" of abstraction. This level describes a circuit in terms of digital signal levels and leaves out most layout information such as geometry. It also leaves out some device details, such as power and ground connections. In the lower part of Figure 4 is a representation of an inverter at the "layout" abstraction level. This inverter is an

implementation of the switches level inverter and it includes many decisions about geometry. In addition, two constraints have been introduced that specify that parts of the layout inverter should be connected to power and ground.

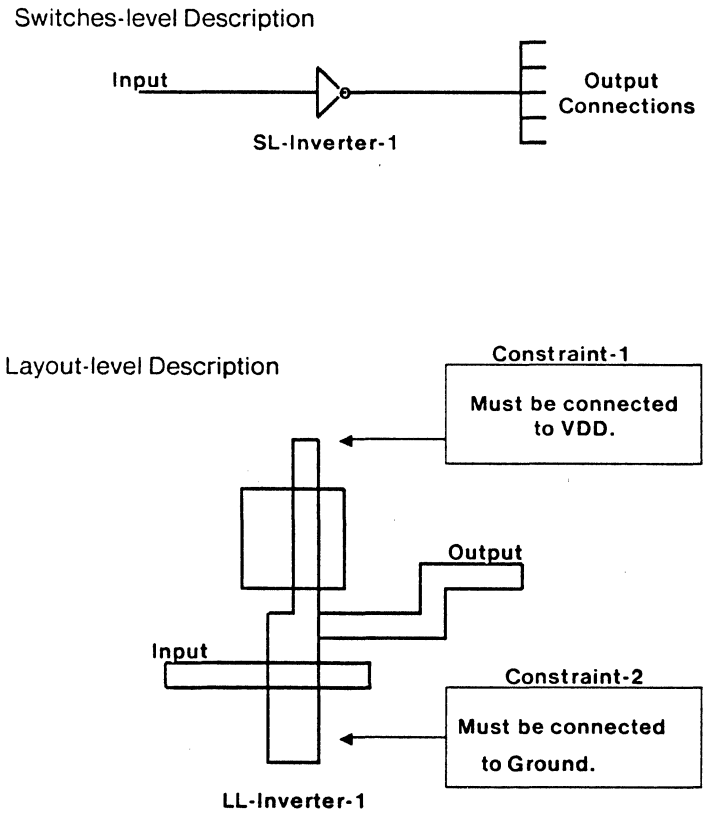


Figure 4. Introducting Constraints in a Hierarchical design. In this figure, the layout level description is an "implementation" of the switches-level description. The designer chooses to introduce constraints about connections to power and ground, and to defer decisions about how to exactly "route" the wires to make the connections.

These constraints can be satisfied by routing appropriate "wires" to other parts of the circuit. If the constraints can not be satisfied, then the inverter will not function and the design will need to be changed, at least at the switches level. The advantage of introducing the two constraints instead of wiring the layout inverter immediately is that this allows decisions about the wiring route to be deferred until later. For example, there may not be enough information yet to determine the location of power and ground. Also there many be other considerations in the design of the circuit more important than this inverter that will bear on the location of power and ground.

The second point illustrates an important phenomenon in problem solving: subproblems interact. Subproblems are said to be *nearly independent* when they can be solved with little (but

not neglible) coordination of the solution process. Typically, these subproblems are locally under-constrained, meaning that they permit more than one solution when only *local* constraints are considered. In this circumstance, it is tempting to act as if the subproblems are independent and to try to solve them separately. In some lucky cases, this works fine but often the difficulties show up when the subproblem solutions are combined. By introducing constraints instead of choosing particular values, the problem solver is able to pursue a *least commitment* style of problem solving. It moves its attention opportunistically between subproblems and avoids overspecifying local decisions.

It is instructive to compare this tentative *posting* of constraints to the making of *tentative* assertions in belief revision systems. Just as belief revision systems can retract beliefs, constraint systems can withdraw or relax constraints. In hypothetical reasoning, assumptions are introduced so that their consequences can be explored. In constraint systems, constraints can be introduced as goals in order to see whether they can be satisfied. For example, a circuit design system may introduce a constraint that the layout area of a component fit within a given rectangle. Introducing the constraint does not assert that there is a design that will satisfy the constraint. The constraint may even turn out to be provably unsatisfiable. Rather, the constraint indicates a tentative subgoal in the search for a satisfactory design. In a belief revision system, the status of an assertion is either *in*, *out*, or *unknown*. In a constraint system, the status may be either *satisfied*, *unsatisfiable*, *withdrawn*, or *proposed*. A constraint is satisfied if values for its arguments have been assigned that make the predicate true. A constraint is unsatisfiable if no set of assignments can be found to satisfy the constraint. A constraint is proposed if it has been formulated, but it is not yet known whether it can be satisfied. A constraint is withdrawn if it is removed from consideration. These characterizations of the possible statuses of a constraint are exemplary; particular constraint systems may employ somewhat different distinctions.

Problem-solving systems that solve problems by constraint satisfaction methods have been discussed in the literature for several years (e.g. Fikes [Fikes70]). For the most part, AI has focused on systems which manipulate algebraic constraints and propagate values (e.g. Sussman [Sussman80]). A formulation of tentative hierarchical reasoning with constraints was demonstrated in an experiment planning system by Stefik [Stefik81a].

2.3.4 *Factoring into meta-Problems*

Problem solvers repeatedly decide what to do next. In the simplest organization, the selection of what to do next is made by a fixed search method. Recently there has been increased interest in systems that can reason *about* their own reasoning processes. In this formulation, there is a *meta-level* problem-solver that chooses among several potential *methods* for deciding what to do next at the object level.

The idea of factoring decisions into meta-problems is illustrated by the following travel planning example. One useful fact about automobiles is:

A car needs gasoline before it can be driven.

In a state-space formulation, the requirement for gasoline would be called a precondition for the "driving" operator. Given this fact, we might expect a driver to check the gasoline supply before starting a trip, and to first plan to get some gasoline as needed. One approach to represent this behavior would be to add the rules:

> **Always check for gasoline before starting a trip.**
> **If there is no gasoline, then get some.**

Unfortunately, there are many similar facts about driving and cars (e.g., rules for battery water, brake fluid, transmission fluid, and oil) and writing such rules for every fact seems redundant. We would like a planner to use more general rules for all of the facts. For example:

> **If a precondition is not satisfied, create a subgoal to satisfy it.**

This rule about testing preconditions is not specific to cars and trips. It is a rule about planning with facts and can be used as part of the planning task. Thus, it is a statement about a meta-problem. It contributes indirectly to the decision about what the driver should do next. This organization suggests an economy of representation by separating problem-solving knowledge from knowledge about the ground domain. Many similar rules about planning are possible, for example:

> **If there are conflicting goals, give them priorities and do the important ones first.**
>
> **If there are subgoals that are part of several major goals, plan to satisfy those subgoals before other subgoals.**

There is no reason to limit consideration to only one meta-level. There can be rules about selecting these planning rules:

> **Keep your options open - avoid planning-rules that limit choices.**

The word *meta-planning* has been used to describe systems that plan about their own planning processes. Stefik [Stefik81b] used a layered meta-planning organization in an expert system (called MOLGEN) that is discussed in Section 4. He suggests that the organization of a problem solver into layers is an important technique for partitioning control considerations. Without this factoring, meta-level interactions can surface as confusing interactions between ground-level tasks. To reduce the apparent complexity of a system, he advocates the use of separate layers for meta-problems until the information in the top-most layer is trivial. Wilensky [Wilensky80] has observed that there is a significant overlap between the knowledge needed for planning and for meta-planning.

Factoring into meta-problems seems to be a useful idea for organizing knowledge in expert problem-solvers, but there has not been much experience with it yet. General proposals for expressing control knowledge in terms of meta-problems have been advanced by de Kleer *et al.* [de Kleer77] and Weyhrauch [Weyhrauch80]. Much more experience in creating such organizations is necessary before general versions of them will be well understood.

2.4 Summary of Basic Concepts

We started our discussion of the basic concepts for building expert systems with symbols and symbol structures. Predicate calculus was reviewed as a formal language for symbol structures. Computers were viewed as active devices that can use symbolic inference to infer new facts from what is known already. Some well-known rules of inference were reviewed. Simply having a set of rules of inference is not enough; we also need knowledge about problem-solving to guide the application of these rules or there will be a combinatorial explosion resulting in the derivation of many correct but irrelevant inferences.

Search was discussed as a formulation of problem-solving with provisions for containing combinatorial explosions. Several search concepts were introduced: goals, search spaces and operators for moving between states. Heuristic search incorporates knowledge about the solution space to guide the search for solutions. This is a general approach to containing combinatorial explosions. The simplest example of a heuristic method is the use of evaluation functions to estimate distance to a goal, but this approach is usually infeasible in expert systems. Heuristics can involve large amounts of symbolic computation. A powerful approach is abstraction of the solution space. Hierarchical generate and test depends on the early use of pruning rules to eliminate classes of solutions.

Practical problem solving requires the making and retraction of assumptions. Examples of this in diagnostic and design problems were given. Belief revision systems keep track of assumptions and inferences by recording justifications for them in a dependency network. Dependency-directed backtracking is an approach to search that is superior to chronological backtracking. Dependency-directed reasoning can be extended to include reasoning about the problem-solving process itself by introducing justifications about goals and constraints. Such reasoning can be organized as a meta-level problem by dividing the goals, actions, and states of the problem solver into meta-levels.

3. A CHARACTERIZATION OF EXPERT TASKS

In this section we will consider several generic tasks that experts perform. Examining these tasks will help us to understand what makes expert reasoning difficult. The difficulties provide a guide to architectural relevance; they enable us to focus on issues that relate to critical steps in reasoning.

Interpretation

Interpretation is the analysis of data to determine their meaning.

Example: Interpretation of mass spectrometer data (Buchanan and Feigenbaum [Buchanan78]). In this case, the data are measurements of the masses of molecular fragments and interpretation means the determination of one or more chemical structures.

Requirements: Find consistent and correct interpretations of the data. It is often important that analysis systems be rigorously complete, that is, that they consider the possible interpretations systematically and discard candidates only when there is enough evidence to rule them out.

Key Problems: Data are often noisy and errorful, that is, data values may be missing, erroneous, or extraneous. (1) This means that interpreters must cope with partial information. (2) For any given problem, the data may seem contradictory. The interpreter must be able to hypothesize which data are believable. (3) When the data are unreliable, the interpretation will also be unreliable. For credibility it is important to identify where information is uncertain or incomplete and where assumptions have been made. (4) Reasoning chains can be long and complicated. It is helpful to be able to explain how the interpretation is supported by the evidence.

Diagnosis

Diagnosis is the process of fault-finding in a system (or determination of a disease state in a living system) based on interpretation of potentially noisy data.

Example: Diagnosis of infectious diseases (Shortliffe [Shortliffe76]).

Requirements: Requirements include those of interpretation. A diagnostician must understand the system organization (i.e., its anatomy) and the relationships and interactions between subsystems.

Key Problems: (1) Faults can sometimes be masked by the symptoms of other faults. Some diagnostic systems ignore this problem by making a *single fault assumption.* (2) Faults can be intermittent. A diagnostician sometimes has to stress a system in order to reveal faults. (3) Diagnostic equipment can itself fail. A diagnostician has to do his best with faulty sensors. (4) Some data about a system can be inaccessible, expensive or dangerous to retrieve. A diagnostician must decide which measurements to take. (5) The anatomy of natural systems such as the human body is not fully understood. A diagnostician may need to combine several (somewhat inconsistent) partial models.

Monitoring

Monitoring means to continuously interpret signals and to set off alarms when intervention is required.

Example: Monitoring a patient using a mechanical breathing device after surgery (Fagan [Fagan80]).

Requirements: A monitoring system is a partial diagnostic system with the requirement that the recognition of alarm conditions be carried out in real time. For credibility, it must avoid false alarms.

Key Problems: What constitutes an alarm condition is often context-dependent. To account for this, monitoring systems have to vary signal expectations with time and situation.

Prediction

Prediction means to forecast the course of the future from a model of the past and present.

Example: Predicting the effects of a change in economic policy. (Some planning programs have a predictive component. There is currently an opportunity to develop expert prediction programs in a variety of areas.)

Requirements: Prediction requires reasoning about time. Predictors must be able to refer to things that change over time and to events that are ordered in time. They must have adequate models of the ways that various actions change the state of the modeled environment over time.

Key Problems: (1) Prediction requires the integration of incomplete information. When information is complete, prediction is not an AI problem (e.g., "where will Jupiter be two years from next Thursday?"). (2) Predictions should account for multiple possible futures (hypothetical reasoning), and should indicate sensitivity to variations in the input data. (3) Predictors must be able to make use of diverse data, since indicators of the future can be found in many places. (4) The predictive theory may need to be contingent; the likelihood of distant futures may depend on nearer but unpredictable events.

Planning

A plan is a program of actions that can be carried out to achieve goals. Planning means to create plans.

Example: Experiment planning in molecular genetics (Stefik [Stefik81]).

Requirements: A planner must construct a plan that achieves goals without consuming excessive resources or violating constraints. If goals conflict, a planner establishes priorities. If planning

requirements or decision data are not fully known or change with time, then a planner must be flexible and opportunistic. Since planning always involves a certain amount of prediction, it has the requirements of that task as well.

Key Problems: (1) Planning problems are sufficiently large and complicated that a planner does not immediately understand all of the consequences of his actions. This means that he must be able to act tentatively, so as to explore possible plans. (2) If the details are overwhelming, he must be able to focus on the most important considerations. (3) In large complex problems, there often are interactions between plans for different subgoals. A planner must attend to these relationships and cope with goal interactions. (4) Often the planning context is only approximately known, so that a planner must operate in the face of uncertainty. This requires preparing for contingencies. (5) If the plan is to be carried out by multiple actors, coordination (e.g. choreography) is required.

Design

Design is the making of specifications to create objects that satisfy particular requirements.

Example: Designing a digital circuit. This is an area of increased interest and activity in expert systems.

Requirements: Design has many of the same requirements as planning.

Key Problems: (1) In large problems, a designer can not immediately assess the consequences of design decisions. He must be able to explore design possibilities tentatively. (2) Constraints on a design come from many sources. Usually there is no comprehensive theory that integrates constraints with design choices. (3) In very large systems, a designer must cope with the system complexity by factoring the design into subproblems. He must also cope with interactions between the subproblems, since they are seldom independent. (4) When a design is large, it is easy to forget the reasons for some design decisions and hard to assess the impact of a change to a part of a design. This suggests that a design system should record justifications for design decisions and be able to use these justifications to explain decisions later. This is especially apparent when subsystems are designed by different designers. (5) When designs are being modified, it is important to be able to reconsider the design possibilities. During redesign, designers need to be able to see the "big picture" in order to escape from points in the design space that are only *locally optimal.* (6) Many design problems require reasoning about spatial relationships. Reasoning about distance, shapes, and contours demands considerable computational resources. We do not yet have good ways to reason approximately or qualitatively about shape and spatial relationships.

Several issues appear repeatedly across this catalog of expert tasks: large solution spaces, tentative reasoning, time-varying data and noisy data. Large solution spaces are typical in interpretation planning and design tasks. In the mass spectrometery example [Buchanan78], some problems require millions of possible chemical structures to be considered. In planning and design tasks, the number of reasonable solutions is usually a very small fraction of a very large number of possible solutions. In each of these tasks, the size and characterization of the solution space is an important organizational parameter. Tentative reasoning is needed in diagnostic, design and planning tasks. Many diagnostic procedures profitably employ assumptions about the number of faults or about the reliability of sensors. Part way through diagnosis, it may be discovered that these assumptions are unwarranted. This places a premium on the ability to undo the effects of the assumptions. Similarly, in design and planning tasks it is often appropriate because of scale to employ simplifying assumptions (e.g., abstractions). In any given design, some of the assumptions will fail in a design, so there is an incentive to employ methods that facilitate the reworking of assumptions and trade-offs during iterations of the design process. Patient monitoring and diagnosis tasks are concerned with situations that evolve over time -- as diseases follow their natural course or as treatments are administered. Finally, sensors often yield noisy data. This is a factor for any task involving reasoning from measurements such as interpretation, diagnostic, and monitoring tasks. The next section considers organizational prescriptions for each of these concerns.

4. KNOWLEDGE ENGINEERING PRESCRIPTIONS

Feigenbaum [Feigenbaum77] defines the activity of *knowledge engineering* as follows:

> "The knowledge engineer practices the art of bringing the principles and tools of AI research to bear on difficult applications problems requiring experts' knowledge for their solution. The technical issues of acquiring this knowledge, representing it, and using it appropriately to construct and explain lines-of-reasoning, are important problems in the design of knowledge-based systems. ... The art of constructing intelligent agents is both part of and an extension of the programming art. It is the art of building complex computer programs that represent and reason with knowledge of the world." [pp. 1014-1016]

This section is intended as a prescriptive guide to building expert systems. To illustrate the strengths and limitations of organizational alternatives we will cite a number of contemporary systems. In presenting these examples we adopt a level of detail that is adequate for making the ideas clear yet removed from the particulars of the task and the programming implementation. The reader seeking a more detailed discussion of implementation is encouraged to consult a textbook on AI programming [Charniak80].

One of the most variable characteristics of expert systems is the way that they search for solutions. The choice of search method is affected by many characteristics of a domain, such as the size of the solution space, errors in the data, and the availability of abstractions. Inference is at the heart of a reasoning system and failure to organize it properly can result in problem-solvers that are hopelessly inefficient, naive, or unreliable. As a consequence of this, search is one of the most studied topics in artificial intelligence.

Our pedagogical style is to start with a very restricted class of problems that admits a very simple search process. We will articulate the domain restrictions under which this organization is applicable, and thereby expose its limitations. Then we will relax the requirements on the task domain and introduce ameliorating techniques as architectural prescriptions. The frontispiece shows a chart of the cases that we will consider. Each box in the figure corresponds to one of the cases and the numbering indicates the order in which the cases are discussed. The lines connecting the boxes organize the cases into a tree structure such that a sequence of cases along a branch corresponds to increasingly elaborate considerations of a basic idea. The first three branches consider the complications of unreliable data or knowledge, time-varying data, and a large search space. Any given problem may require combining ideas from any of these topics. The problem of a large search space is then considered along three major branches. The first branch (cases 5 through 8) considers organizations for abstracting a search space. The second branch focuses on methods for incomplete search. The third branch considers only ways to make the knowledge base itself more efficient. This breakdown is mainly pedagogical. Real systems may combine these ideas.

4.1 Case 1 -- Small Search Space with Reliable Knowledge and Data

Systems for complex tasks are generally more complicated than systems for simple tasks. In this section we will consider a very simple architecture which has been used for relatively simple applications. We begin by listing the requirements for task simplicity:

1. The data and knowledge should be reliable.
2. The data and knowledge should be static.
3. The space of possible solutions should be small.

On the surface these requirements seem quite mild. Indeed, there is a widely held belief among people who have not looked closely into problem solving that most problems satisfy these requirements. After all, for many problems the facts seem straightforward and there are not *that* many solutions to consider. Under closer examination, however, most real tasks fail to meet these requirements including the examples of expert tasks listed in Section 3.

The first requirement is that the data and knowledge be reliable. Reliable *data* are not noisy or errorful. There can be no extraneous signals and no missed signals (e.g., due to sampling). In real applications few sources of data meet these requirements. In addition to data reliability, the knowledge must be reliable. Reliable knowledge is applicable without concern about consistency or correctness. Systematic application of reliable knowledge should not lead to false, approximate, or tentative conclusions. The main advantage of reliability for both data and knowledge is the monotonicity of the system. In the simplest architecture, the memory is a monotonic data base to which conclusions are added by the reasoning system. No provisions need to be made for retraction of facts given new information. It is enough to develop a single line of reasoning; that is, there is no need to develop multiple arguments to support potential conclusions. If more than one inference rule is applicable at a given time, the order in which they are applied is unimportant.

The second requirement is intended to avoid the problem of reasoning with time-dependent data. This means that the system need not be concerned with invalidating facts as time passes.

The requirement that the search space should be small implies that no provisions need to be made to cope with the limitations of computational resources. There is to be no concern about computationally efficient data structures or avoidance of combinatorial explosions. It doesn't matter whether the search is for one solution or for all possible solutions as long as the space is small. If the search is exhaustive, the maximum size of the solution space depends on the time it takes to consider a single solution. A useful number to keep in mind for this maximum is ten factorial (10!). If 25 milliseconds are required to consider a solution, then 10! solutions can be considered sequentially during a full twenty-four hour day. This provides a practical upper limit for exhaustive search. The surprise is that the ceiling is so low.

An organization for solving these problems has two main parts: a memory and an inference method. The simplest organization of the memory would be a list of inferred facts (i.e., beliefs). For many problems, the beliefs can be limited to the predicate calculus such as:

(On Block1 Block2)
(NOT (On Block2 Table-1)) .

Some systems optimize the storage format of the data. For example, in frame systems (Bobrow and Winograd [Bobrow77]) the indexing of the wffs is organized to make the most common access paths more efficient. Data which are used together are stored together in frames. In the following sections we relax these restrictions on the problems.

4.2 Case 2 -- Unreliable Data or Knowledge

Experts sometimes make judgments under pressure of time. All of the data may not be available; some of the data may be suspect; some of the knowledge for interpreting the data may be unreliable. These difficulties are part of the normal state of affairs in many interpretation and diagnostic tasks. The problem of drawing inferences from uncertain or incomplete data has invited a variety of technical approaches.

One of the earliest and simplest approaches to reasoning with uncertainty was incorporated in the MYCIN expert system (Davis [Davis77], Shortliffe [Shortliffe76]) for selecting antibiotic therapy for bacteremia. One of the requirements for MYCIN was to represent judgmental reasoning such as "A *suggests* B" or "C and D *tend to* rule out E". To this end, MYCIN introduced a model of approximate implication using numbers called *certainty factors* to indicate the strength of a heuristic rule. The following is an example of a rule from MYCIN's knowledge base:

If the infection is primary-bacteremia, and
the site of the culture is one of the sterile sites, and
the suspected portal of entry of the organism
is the gastro-intestinal tract
then there is suggestive evidence (.7) that the identity of the organism is bacteroides.

The number ".7" in this rule indicates that the evidence is strongly indicative (.7 out of 1) but not absolutely certain. Evidence confirming a hypothesis is collected separately from that which disconfirms it and the "truth" of the hypothesis at any time is the algebraic sum of the evidence. This admits the combination of evidence in favor and against the same hypothesis.

The introduction of these numbers is a departure from the *exactness* of predicate calculus. In MYCIN, things are not just true or false; reasoning is inexact and that inexactness is numerically characterized in the rules by an expert physician. Facts about the world are represented as 4-tuples with numeric truth values. For example:

(IDENTITY ORGANISM-2 KLEBSIELLA .25)

means that "The identify of organism-2 is *Klebsiella* with certainty .25." In predicate calculus, the rules of inference tell us how to combine wffs and truth values. MYCIN has its own way to combine formulas. When the premise of a rule is evaluated, each predicate returns a number between 1 and -1 (-1 means "definitely false"). MYCIN's version of **and** performs a minimization of its arguments; **or** performs a maximization of its arguments. This results in a numerical value between -1 and 1 for the premise of a rule. For rules whose premise values surpass an empirical threshold of .2, the rule's conclusion is made with a certainty that is the product of the premise value and the certainty factor of the rule. These rules of combination can be shown to have certain

properties -- such as insensitivity to the order in which the rules are applied. MYCIN's certainty factors are derived from probabilities but have some distinct differences [Shortliffe76].

A reasonable question about such approaches is whether they are unnecessarily *ad hoc.* A commonly voiced criticism is that MYCIN introduces its own formalism for reasoning with uncertainty when there are thoroughly-studied probabilistic approaches available. For example, Bayes' Rule could be used to calculate the probability (e.g., of a disease) in light of specified evidence, from the *a priori* probability of the disease and the conditional probabilities relating the observations to the diseases. The main difficulty with Bayes' Rule is the large amount of data that are required to determine the conditional probabilities needed in the formula. This amount of data is so unwieldy that the conditional independence of observations is often assumed. It can be argued that such independence assumptions undermine the rigorous statistical model. A middle ground which replaces observations with subjective estimates of prior probabilities has been proposed by Duda, Hart, and Nilsson [Duda76] and analyzed for its limitations by Pednault, Zucker, and Muresan [Pednault81].

Another approach to inexact reasoning that diverges from classical logic is fuzzy logic as discussed by Zadeh [Zadeh79a] and others. In fuzzy logic, a statement like "X is a large number" is interpreted as having an imprecise denotation characterized by a *fuzzy set.* A fuzzy set is a set of values with corresponding *possibility values* as follows:

Fuzzy Proposition:
X is a large number.

Corresponding Fuzzy Set:
(X $\in$ [0,10], .1)
(X $\in$ [10,1000], .2)
({X > 1000}, .7)

The interpretation of the proposition "X is large" is that "X might be less than 10" with possibility .1, or between 10 and 1000 with possibility .2, and so on. The fuzzy values are intended to characterize an imprecise denotation of the proposition.

Fuzzy logic deals with the laws of inference for fuzzy sets. Its utility in reasoning about unreliable data depends on the appropriateness of interpreting *soft data* (see Zadeh [Zadeh79b]) as fuzzy propositions. There is little agreement among AI researchers on the utility of these *modified logics* for intelligent systems, or even on their advantages for reasoning with incomplete data.

The pseudo-probability and fuzzy approaches for reasoning with partial and unreliable data depart from the predicate calculus by introducing a notion of inexactness. Other approaches are possible. For example, the belief revision work cited in the previous section attempts to cope with partial information while preserving the *exactness* of the predicate calculus. Belief revision systems admit precise statements about incomplete information. Any "fuzziness" enters only as non-monotonicity when we consider the system as a whole.

The use of *exact* inference methods on unreliable data in an expert system is illustrated by the GA1 program reported by Stefik [Stefik78]. GA1 is a data interpretation system which copes with errorful data. It exploits the redundancy of experimental data in order to correct errors that it may contain.

GA1 infers DNA structures from segmentation data. GA1's task is to assemble models of complete DNA structures given data about pieces (called segments) of the structures. The segment data are produced by chemical processes involving enzymes that break DNA apart in predictable ways. In a typical problem, several independent breaking processes called digestions are performed and the resulting segments are measured. These digestions give independent measurements of the DNA molecule. For example, independent estimates of the molecular weight can be computed by summing the weights of the segments in any of the "complete" digestions. A digest is called complete if all of the molecules have been cleaved in all possible places by the enzymes.

An example of a rule for correcting missing data is:

> **If a segment appears in a complete digestion for an enzyme that fails to appear in the incomplete digestion for that enzyme, it may be added to the list of segments for the incomplete digestion.**

This rule is based on the observation that segments are easier to overlook in incomplete digestions than in complete digestions. It places more confidence in data from complete digestions than from incomplete ones. Other data correction rules incorporate more elaborate reasoning by modeling predictable instrument error such as failure to resolve measurements which are too close together. Such rules enable GA1 to look to the data for evidence of instrument failure.

All of the data correction in GA1 could have been modeled by a belief revision system. Rules like that above could be used to justify beliefs about which data are correct. GA1 could then have begun its interpretation task and withdrawn any of its interpretations when data were invalidated. GA1 used a much simpler approach since all of the data correction could be done at once by running the correction rules in a fixed order. Then GA1 interpreted the data by searching a large space of possible solutions using a method described in Section 4.4. Since it was possible to correct all of the data before interpretation, it was not necessary to be able to retract interpretations of the data, and the machinery for recording and revising justifications was not needed.

In summary, several methods for reasoning with unreliable data and knowledge have been proposed. The probability-related and possibility methods use modified logics to handle approximations. They use numerical measures for combining evidence. In contrast, data correction rules can reason with partial information without compromising the exactness of predicate calculus. If it is necessary to reason from the data before correcting it, such rules can be incorporated into belief revision systems. All of these methods depend on the formalization of extra *meta-knowledge* in order to correct the data, take back assumptions, or combine evidence. The availability of this meta-knowledge is a critical factor in the viability of these approaches to particular applications. A special method for contending with both fallible knowledge and limited computational resources will be considered in Section 4.10.

4.3 Case 3 -- Time-varying Data

Some expert tasks involve reasoning about situations that change over time. One of the earliest approaches in AI to take this into account was the situational calculus introduced by McCarthy and Hayes for representing sequences of actions and their effects [McCarthy69]. The central idea is to include "situations" along with the other objects modeled in the domain. For example, the formula

(ON Block-1 Table-2 Situation-2)

could represent the fact that in Situation-2, Block-1 is on Table-2. A key feature of this formulation is that situations are discrete. This discreteness reflects the intended use of this calculus in robot planning problems. A robot starts in an initial situation and performs a sequence of actions. After each action, the state of the robot's world is modeled by another situation. In this representation, a situation variable can take situations as values. In some implementations, the actual situation variable is left implicit by indexing the formulas according to situations.

Actions in the situational calculus are represented by functions whose domains and ranges are situations. For each action, a set of frame axioms characterizes the set of assertions (i.e., "the frame") that remains fixed while an action takes place within it. In a robot planning task, an example of an action would be the *Move* action for moving an object to a new location. A frame axiom for *Move* would be that all objects not explicitly moved are left in their original location.

While several issues can be raised about this approach to representing changing situations, AI systems have used it for a variety of tasks with only minor variations. Sometimes the changes of situation are signalled by time-varying data, rather than by the autonomous actions of a robot. An example of this is shown by the VM system reported by Fagan [Fagan79, Fagan80]. VM (for Ventilator Manager) is a program that interprets the clinical significance of patient data from a physiological monitoring system. VM monitors the post-surgical progress of a patient requiring mechanical breathing assistance. A device called a mechanical ventilator provides breathing assistance for seriously ill patients. The type and settings of the ventilator are adjusted to match the patient's needs. As the patient's status improves, various adjustments and changes are made, such as replacing the mechanical ventilator with a "t-piece" to supply oxygen to the patient. In VM's application, a patient's situation is affected by the progression of disease and the response to therapeutic interventions. For such applications, the model of clinical reasoning must account for information received from tests and observations over time.

VM illustrates knowledge suitable for coping with time-varying data. This knowledge in VM is organized in terms of several kinds of rules: transition rules, initialization rules, status rules, and therapy rules. Periodically VM receives a new set of instrument measurements and then it reruns all of its rules. Transition rules are used to detect when the patient's state has changed, e.g., when the patient starts to breathe on the T-piece. The following is an example of a transition rule:

If the current context is "Assist" and
the respiration rate has been stable for 20 minutes and
the I/E ratio has been stable for 20 minutes
then the patient is on "CMV" (controlled mandatory ventilation).

This rule governs the transition between an "Assist" context and a "CMV" context. When the premise of a transition rule is satisfied in VM, a new "context" is entered. These contexts correspond to specific states or situations. When a context is changed, VM uses initialization rules to update its information for the new context (e.g., expectations and unacceptable limits for the measurements). These rules refer to the recent history of the patient to establish new expectations and information for the new context. Part of one such rule follows:

If the patient transitioned from "Assist" to "T-piece" or
the patient transitioned from "CMV" to "T-piece"
then expect the following:

		[	-- Acceptable --		]	
	very low	low	[min	- ideal - max]	high	very high
SYS		110			150	
DIA		60			95	
MAP	60	75			110	120
Pulse rate		60			120	
ECO2	22	28	30	40	45	50
...						

VM's reasoning about time is limited to adjacent time intervals. It is concerned only with the previous state and the next state. Its mechanisms for dealing with this are state-triggered expectations and rules for dynamic belief revision. The transition rules in VM govern changes of context. Data arrives periodically, but context is changed only when it is adequately supported by the evidence. The initialization rules are essentially like the frame axioms -- establishing what changes and what stays the same in the new context. Once a context is set, the expectations are used to govern VM's behavior until the context is changed again.

Programs which need to reason about more distant events require more elaborate representations of events and time. For example, planning and prediction tasks require reasoning about possible futures. For these applications, the situational calculus must be extended to allow for multiple possible futures with undetermined operations, unordered sets of possible future events, and the possible actions of uncontrolled multiple actors. While AI systems capable of such sorts of reasoning seem within reach, their construction is still a research enterprise.

4.4 Case 4 -- Large but Factorable Solution Space

In the restricted class of problems in Section 4.1, it was stipulated that the data and knowledge must be reliable, the data must be static and the search space must be small. We have already discussed some techniques for relaxing the first two requirements. In this section we will begin the consideration of techniques for coping with very large search spaces.

In many data analysis tasks, it is often desirable to find every interpretation that is consistent with the data. This conservative attitude is standard in high risk applications such as the analysis of poisonous substances or medical diagnosis. A systematic approach would be to consider all possible cases, and to rule out those that are inconsistent with the data. This approach is called *reasoning by elimination* and has been familiar to philosophers for years, but it has often been regarded as impractical. The difficulty is that there is often no practical way to consider all of the possible solutions.

The DENDRAL program [Buchanan78] is probably the best known AI program that reasons by elimination (using generate and test). The key to making it work is to incorporate early pruning into the generate and test cycle as suggested in Section 3.2.5. In this section we will examine this method in more detail, illustrate some of the characteristics of problems on which it will work, and give examples of the kinds of knowledge that can be incorporated to make this method practical. Since the problem area for the DENDRAL program is rather complicated for tutorial purposes, we will consider the simpler expert system, GA1 [Stefik78], that was already mentioned in Section 4.2.

Like DENDRAL, GA1 is a data interpretation program that infers a complete molecular structure from measurements of molecular pieces. Figure 5 shows a simple example of the kind of data that GA1 would have about a molecule. The top part of the figure shows that a molecule is made up of segments of measureable length. The lines labeled *A* and *B* that cut across the circle indicate the sites where the molecule is cleaved by enzymes (named A and B, respectively). All of the molecules that GA1 is concerned with are linear or circular. This means that all of the molecular segments are linear pieces that can be arranged end to end. When a sample of molecules is completely digested by an enzyme, pieces are released whose sizes can be measured. The goal is to infer the structure of the original molecule from the digest data. Sometimes more than one molecular structure is consistent with the available data.

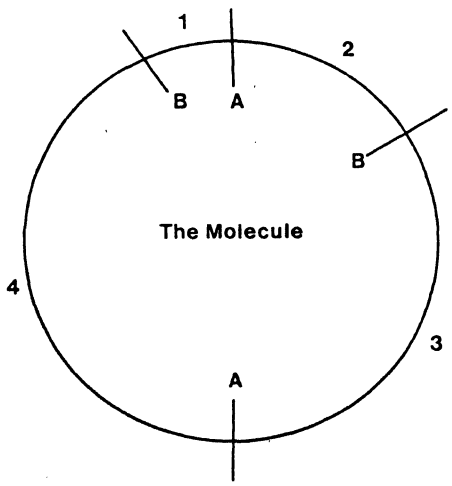
A primary task in problems like this is to create a workable generator of all of the possible solutions (i.e., all of the possible molecules). In GA1, the first step is to apply data correction rules as shown in Section 4.2. Then GA1 determines an initial set of generator constraints -- a set of segments and enzyme sites for building candidate molecules. The rules for deriving the list will not be elaborated here, but they make conservative use of molecular weight estimates and redundant data from several digests. The generation process then begins by combining these segments and sites, and testing whether the combinations fit the evidence. For example, the following lists (among others) correspond to the complete molecule in Figure 5:

(1 A 2 B 3 A 4 B)
(2 B 3 A 4 B 1 A)
(1 B 4 A 3 B 2 A)

These equivalent representations can be generated by starting with any of the four segments in the picture of the molecule in Figure 5, and reading off the sites and segments around the circle either clockwise or counterclockwise. This provides us with eight equivalent representations for the same molecule. The situation is analogous to the case where English sentences have different syntax but

the same meaning. A generator is said to be nonredundant if it produces exactly one of the equivalent representations of a solution (the *canonical form*) during the generation process. GA1 does this by incorporating rules for pruning non-canonical structures during the generation process. An example of such a rule follows:

> **If circular structures are being generated, only the smallest segment in the list of initial segments should be used for the first segment.**



The Data

Complete A Digest: 5 5
Complete B Digest: 3 7
Complete A&B Digest: 1 2 3 4

Figure 5. Enzyme A cleaves the cirvular molecule at the points labeled A. Enzyme B cleaves it at the points labeled B. The table lists the fragments that would be observed under ideal digestion experiments.

The key to effective use of generate and test is to prune classes of inconsistent candidates as early as possible. For example, consider the following structure from the generation process for the sample problem:

(1 B 2 B).

GA1 treats this as a description of all of the molecules that match the pattern:

(1 B 2 B <Site> <Site>)

where any of the remaining segments and sites may be filled in the template. It is easy to see that no molecule matching this template is consistent with the data in Figure 5 -- because all such molecules would yield a segment of length 2 in the complete digest for B. Other pruning considerations are more global. When such pruning rules exist, the solution space is said to be factorable.

GA1 has been run on problems where the number of possible candidates is several billion. However, the pruning rules are so effective at eliminating classes of solutions that most problems require only a few seconds of computer time.

After the generator is finished, usually twenty or thirty candidates make it through all of the pruning rules. Only one or two of these will be consistent with all of the data. In principle, it is possible to add more pruning rules to GA1 so that only the consistent solutions remain. However, the rules needed to do this become increasingly complex and specialized. It becomes difficult to prove the correctness of such rules (so that solutions will not be missed) and to ensure that the rules are faithfully represented in the program. There is also a point of diminishing returns when each new specialized rule covers a smaller number of cases. In GA1 this problem is addressed by applying a digestion process model to the final candidates and comparing its predictions with the observed data. A simple scoring function then penalizes candidates that predict extra or missing segments. Because the number of disagreements between the idealized digests of two *different* molecules diverges rapidly for small molecular differences, it was not necessary to tune the scoring function to recognize wrong solutions.

In summary, generate-and-test is an appropriate method to consider when it is important to find all of the solutions to a problem. For the method to be workable, however, the generator must partition the solution space in ways that allow for early pruning. These criteria are often satisfied in data interpretation and diagnostic problems.

4.5 Case 5 -- No Evaluator for Partial Solutions

There are many problems involving large search spaces for which generate and test is a method of last resort. The most common difficulty is that no generator of solutions can be found for which early pruning is viable. Design and planning problems are of this nature. One usually cannot tell from a fragment of a plan or design whether that fragment is part of a satisfactory complete solution; there is no reliable evaluator of partial solutions *expressed as solution fragments.*

In this section we will consider the first of several approaches to problem solving without early pruning. These approaches have in common the idea of abstracting the search space (see Section 3.2.4), but differ in their assumptions about the nature of that space. Abstraction emphasizes the important considerations of a problem and enables its partitioning into subproblems. In the simplest case there is a fixed partitioning in the abstract space which is appropriate for all of the problems in an application.

This case is illustrated by the R1 program reported by McDermott [McDermott80]. R1's area of expertise is the configuring of Digital Equipment Corporation's VAX computer systems. Its input is a customer's order and its output is a set of diagrams displaying the spatial relationships among the components on the order. This task includes a substantial element of design. In order to determine whether a customer's order is satisfactory, R1 must determine a spatial configuration for the components and add any necessary components that are missing.

The configuration task can be viewed as a hierarchy of subtasks with strong temporal interdependencies. R1 partitions the configuration task into six ordered subtasks as follows:

1. **Determine whether there is anything grossly wrong with the customer's purchase order (e.g., mismatched items, major prerequisites missing).**
2. **Put the appropriate components in the cpu and the cpu expansion cabinets.**
3. **Put boxes in the unibus expansion cabinet and put the appropriate components in those boxes.**
4. **Put panels in the unibus expansion cabinets.**
5. **Lay out the system on the floor.**
6. **Do the cabling.**

The actions within each subtask are highly variable; they depend on the particular combination of components in an order and on the way these components have been configured so far. Associated with each subtask in R1 is a set of rules for carrying out the subtask. An example of a rule for the third subtask follows:

If the most current active context is assigning a power supply
and a unibus adaptor has been put in a cabinet
and the position it occupies in the cabinet (its nexus) is known
and there is space available in the cabinet for a power supply for that nexus
and there is an available power supply
and there is no H7101 regulator available
then add an H7101 regulator to the order.

R1 has about 800 rules about configuring VAX systems. Most of the rules are like the example above. They define situations in which some partial configuration should be extended in particular ways. These rules enable R1 to combine partial configurations to form an acceptable configuration. They indicate what components can (or must) be combined and what constraints must be satisfied in order for the combinations to be acceptable. They make use of a data base describing properties of about 400 VAX components. Other rules describe the temporal relationships between subtasks by determining their ordering. (These rules are analogous to the transition rules described for VM in Section 5.3 except that the rules monitor the state of R1's problem solving instead of data from external sensors.)

The approach that R1 uses is called *Match.* It is one of Newell's *weak methods* for search [Newell73]. Match enables R1 to explore the space of possible configurations with the basic operations of creating and extending partial configurations. Match explores this space by starting in an initial state, going through intermediate states, and stopping in a final state *without any backtracking.* Each state in the space is a partially instantiated configuration. R1 proceeds through its six major tasks in the same order for each problem; it never varies the order and it never backs up in any problem. The benefit of the abstraction space is that R1 needs to do very little search.

The conditions that make Match viable are both its source of power and its weakness. The key requirement is that there can be no backtracking. This means that at any intermediate state, R1 must be able to determine whether the state is on a solution path. This requires that there must exist a partial ordering on decisions for the task such that the consequences of applying an operator bear only on "later" parts of the solution.

Match is insufficient for the complete task in R1. The subtask of placing modules on the unibus is formulated essentially as a bin-packing problem -- namely how to find an optimal sequence that fits within spatial and power constraints. No way of solving this problem without search is known. Consequently R1 uses a different method for this part of the problem.

In summary, abstraction should be considered for applications where there is a large search space but no method for early pruning. R1 is an example of a system which uses a fixed abstract solution. Within this framework, it uses the Match method to search for a solution. Whether Match is practical for an application depends on how difficult it is to order the intermediate states. They must be ordered so that no state can be reached before all the required information for deciding whether that state is on a solution path has been determined.

4.6 Case 6 -- No Fixed Partitioning of Subproblems

When every example problem in an application can be usefully partitioned into the same subproblems, then the organization described in the previous section should be considered. In applications with more variety to the problems, no fixed set of subproblems can provide a useful abstraction. For example, planning domains such as errand running [Hayes-Roth79] require plans rich with structure. To be useful, abstractions must embody the variable structure of the plans.

In this section we will consider an approach called *top-down refinement* that tailors an abstraction to fit each problem. The following aspects of the approach are important:

1. Abstractions for each problem are composed from terms (selected from a space of terms) to fit the structure of the problem.
2. During the problem-solving process, these concepts represent partial solutions that are combined and evaluated.
3. The concepts are assigned fixed and predetermined abstraction levels.
4. The problem solution proceeds *top down*, that is, from the most abstract to the most specific.
5. Solutions to the problem are completed at one level before moving down to the next more specific level.
6. Within each level, subproblems are solved in a problem-independent order. (This creates a *partial* ordering on the intermediate abstract states.)

The best known example of a program using this approach is the ABSTRIPS program reported by Sacerdoti [Sacerdoti74]. ABSTRIPS was an early robot planning program. It made plans for a robot to move objects (e.g., boxes) between rooms. A design goal for ABSTRIPS was to provide abstractions

sufficiently different from the detailed "ground" space to achieve a significant improvement in problem-solving efficiency, but sufficiently similar so that the mapping down from abstractions would not be complex or time-consuming. This led to an interesting and simple approach for representing abstractions.

Abstractions in ABSTRIPS are plans. They differ from ground level plans only in the level of detail used to specify the preconditions of operators. This level of detail is indicated by associating a number (termed a *criticality value*) with all of the literals used in preconditions. For example, Sacerdoti suggested the following criticality assignments in a robot planning domain:

Type and Color	**4**
InRoom	**3**
PluggedIn and Unplugged	**2**
NextTo	**1**

In this example, the predicates for *Type* and *Color* of objects are given high criticalities, since the robot has no operator for changing them. These predicates together with the set of robot actions are combined to form plans for solving particular problems; the space of possible plans is the set of all of the plans that can be built up from these pieces. The most abstract plans are the ones that include only the higher criticality concepts.

Planning in ABSTRIPS starts by setting criticality to a maximum. Planning within each level proceeds backwards from goals. Preconditions whose criticality is below the current level are invisible to the planner, since it is presumed that they will be accounted for during a later pass. After a plan is completed at one level, the criticality level is decremented and planning is started on the next lower level. The previous abstract version of the plan is used to guide the elaboration of the plan. For example, an early version of a plan may determine the route that the robot takes through the rooms. In more detailed versions, steps for opening and closing doors are included. In this way, the abstract plans converge to the specific plan. The sequence of abstract plans is created differently for each problem.

ABSTRIPS was a great advance over its predecessor STRIPS, which lacked the hierarchical planning ability. Generally, when hierarchical and non-hierarchical approaches have been systematically compared, the former have dominated. ABSTRIPS was substantially more efficient than STRIPS, and the effect increased dramatically as longer plans were tried. Since then, many other hierarchical planning programs have been created. In most of the later programs, the abstraction concepts have been arranged in a hierarchy, without actually assigning them criticality numbers.

In summary, the interesting feature of top-down refinement is the flexibility of the abstractions. Abstraction states are individually constructed to fit each problem in the domain. In contrast to Match, top-down refinement places only a *partial* ordering on the intermediate states of the problem-solver. Still, there are some important conditions about problem solving in the domain of application that are inherent in the method. The basic assumption is that a small fixed amount of problem solving knowledge about criticality levels and top-down generation is sufficient.

Furthermore, it must be possible to assign a partial criticality ordering to the domain concepts. What is important for one problem must be important for all of the problems.

4.7 Case 7 -- Interacting Subproblems

One basic difficulty with top-down refinement is the lack of feedback from the problem-solving process. It is presumed that the same kinds of decisions should be made at the same point (i.e., criticality level) for each problem in the domain. In this section, we will explore an approach that is based on a different principle for guiding the reasoning process called the *least commitment* principle. The basic idea is that decisions should not be made arbitrarily or prematurely. They should be postponed until there is enough information.

Reasoning based on the least-commitment principle requires the following abilities:

1. The ability to know when there is enough information to make a decision.
2. The ability to suspend problem-solving activity on a subproblem when the information is not available.
3. The ability to move between subproblems, restarting work as information becomes available.
4. The ability to combine information from different subproblems.

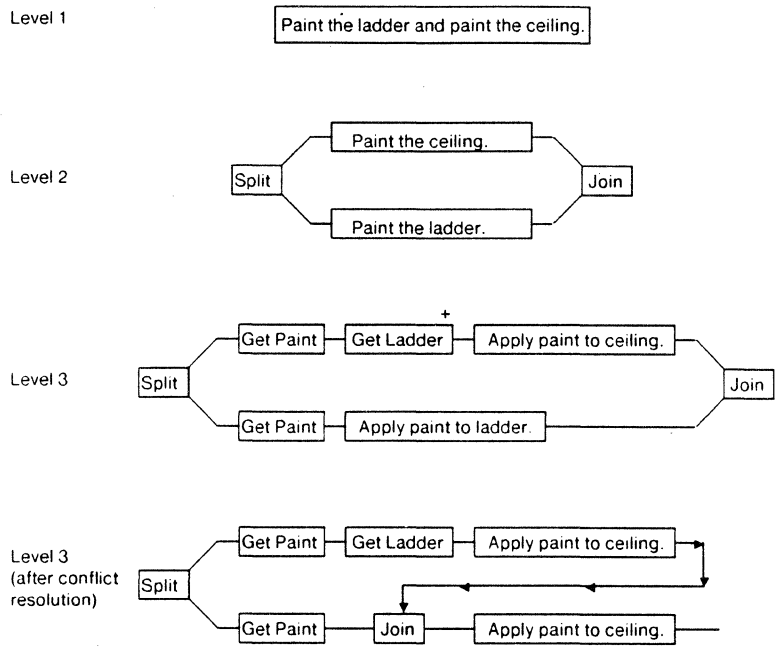


Figure 6. Example of planning in NOAH. NOAH analyses the interactions between the steps in order to assign them an order in time. In this example, the "painting the ladder" is seen to be in conflict with using it. To complete both goals, NOAH decides to paint the ceiling first. Later processing will factor out common subplans like "get paint".

Figure 6 shows an example of this style of reasoning from the NOAH system reported by Sacerdoti [Sacerdoti77]. NOAH was a robot planning system that used a least-commitment approach to assign a time-ordering to operators in a plan. Earlier planning programs inserted operators into a plan as they worked backwards from goals. In contrast, NOAH assigned the operators only a *partial* ordering and added specifications for a complete ordering of the operators only as required.

In Figure 6, NOAH starts with two subgoals: paint the ceiling and paint the ladder. Plans for the two subgoals are expanded in parallel and a conflict is found. If the ladder is painted first, it will be wet and we won't be able to paint the ceiling. In other words, the step to paint the ladder violates a precondition (that the ladder be usable) for the step to paint the ceiling. This interference between the subgoals provides NOAH with enough information to order the tasks. If it had arbitrarily ordered the steps and painted the ladder first, it would have had to plan around the wet ladder, perhaps waiting for it to dry. The resulting plan would not have been optimal.

Another example of this style of reasoning was used in MOLGEN reported by Stefik [Stefik81a]. MOLGEN is an expert system for designing molecular genetics experiments. MOLGEN's architecture involves the following features:

1. Interactions between subproblems are represented as constraints.
2. Interactions between subproblems are discovered via constraint propagation.
3. MOLGEN uses explicit meta-level *problem-solving operators* (as opposed to domain-specific operations) to reason with constraints. (See Figure 7.)
4. MOLGEN alternates between least commitment and heuristic strategies in problem-solving.

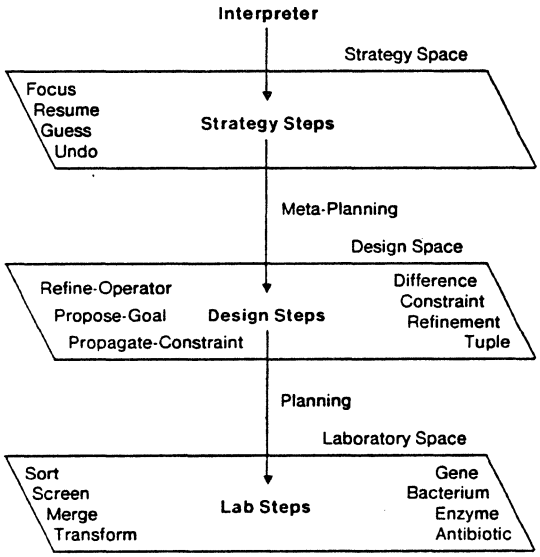


Figure 7. MOLGEN's planning spaces. The design space *plans* by selecting and executing laboratory steps; the strategy space *meta-plans* by selecting and executing design steps.

In the least commitment strategy, MOLGEN makes a choice only when its available constraints have sufficiently narrowed its alternatives. Its problem-solving operators are capable of being suspended so that a decision can be postponed. Constraint propagation is the mechanism for moving information between subproblems. It enables MOLGEN to exploit the synergy between decisions in different subproblems. In contrast with ABSTRIPS' strict backward expansion of plans within levels, MOLGEN expands plans opportunistically in response to the propagation of constraints.

The fourth feature illustrates an interesting limitation of the least commitment principle. Every problem-solver has only partial knowledge about solving problems in a domain. With only the least-commitment principle, the solution process must come to a halt whenever there are choices to be made, but no compelling reason for deciding any of them. We call this situation a *least-commitment deadlock* because operations in all of the subproblems go into a "waiting for more constraints" state. When MOLGEN recognizes this situation, it switches to its heuristic approach and makes a guess. (See Figure 8.) In many cases, a guess will be workable, and the solution process can continue to completion. In other cases, a bad guess can lead to conflicts. The number of conflicts caused by (inaccurate) guessing is a measure of the incompleteness of the problem-solving knowledge. Conflicts can also arise from the least-commitment process in cases where the goals are fundamentally unattainable.

In summary, the least commitment principle coordinates decision-making with the availability of information and moves the focus of problem-solving activity among the available subproblems. The least commitment principle is of no help when there are many options and no compelling reasons for choices. In these cases, some form of plausible reasoning is necessary. In general, this approach uses more information to control the problem-solving process than the top-down refinement approach.

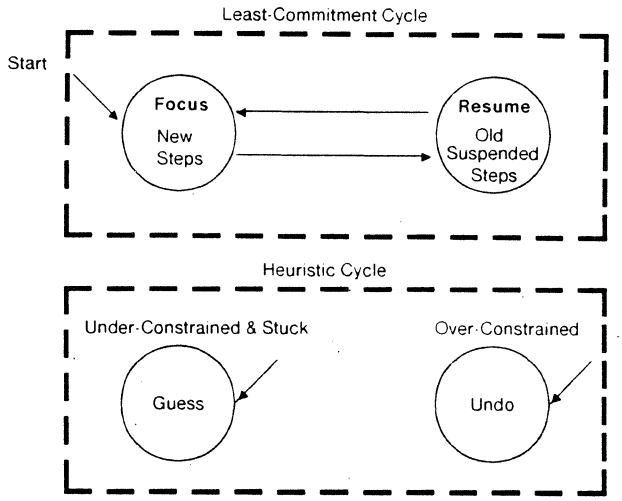


Figure 8. Least-commitment and heuristic cycles. This diagram shows how the strategy operators are controlled by MOLGEN's interpreter. The least-commitment cycle makes conservative changes in the plan, depending on synergy between subproblems (and constraint propagation) to keep going. When MOLGEN runs out of least-commitment steps, it resorts to guessing, using the heuristic cycle.

4.8 Case 8 -- Guessing is Needed

Guessing or *plausible reasoning* is an inherent part of heuristic search. For example, the generator in a generate and test system guesses about partial solutions so that they can be tested. Another example is problem-solvers based on top-down refinement. These problem-solvers implicitly assume that they will be able to refine higher abstractions to specific solutions. Some examples follow of generic situations where guessing is important:

1. Many problem-solvers need to cope with incomplete knowledge and may be unable to determine the best choice at some stage in its problem solving. In such cases, a problem-solver is unable to finish without making a guess. Examples of this are assumptions introduced as a first step in hypothetical reasoning and assumptions introduced to break a least commitment deadlock.

2. A search space may be quite dense in solutions. If solutions are plentiful and equally desirable and there is no need to get them all, guessing can be efficient.

3. Sometimes there is an effective way to *converge* to solutions by systematically improving approximations. (Top-down refinement is an example of this.) In cases where convergence is rapid, it may be appropriate to guess even when solutions are rare.

The difficulty with guessing is in identifying wrong guesses and recovering from them efficiently. This section considers how plausible reasoning can benefit from partidular architectural features.

Forward reasoning with electrical laws is used to compute electrical parameters (e.g., voltage or current) at one node of a circuit from parameters at other nodes. EL uses only a few laws, such as Ohm's Law, which defines a linear relationship between voltage and current for a resistor, and Kirchoff's current law, which states that the current flowing out of a node equals the current flowing into it. Much of EL's power derives from the exhaustive application of these laws and the ability to reason with these laws symbolically as shown in Figure 9.

This figure illustrates a circuit in which resistors are connected in a ladder arrangement. The analysis task is to determine the voltages and resistances at all of the nodes of the circuit. Symbolic reasoning about the circuit is much simpler than writing and solving equations for the series and parallel resistor network. Analysis begins with the introduction of a variable e to represent the unknown node voltage at the end of the ladder. This yields a current $e/5$ through resistor R_6. Then by Kirchoff's Law, we have the same current through R_5 which gives us a voltage 2_e on the left of the resistor. This voltage across R_4 allows us to compute the current through it using Ohm's Law. The application of electrical laws in terms of symbolic unknowns continues until all of the voltages are defined in terms of e. Finally we have

$$\mathbf{8e = 10\ volts}$$
$$\mathbf{e = 5/4\ volts.}$$

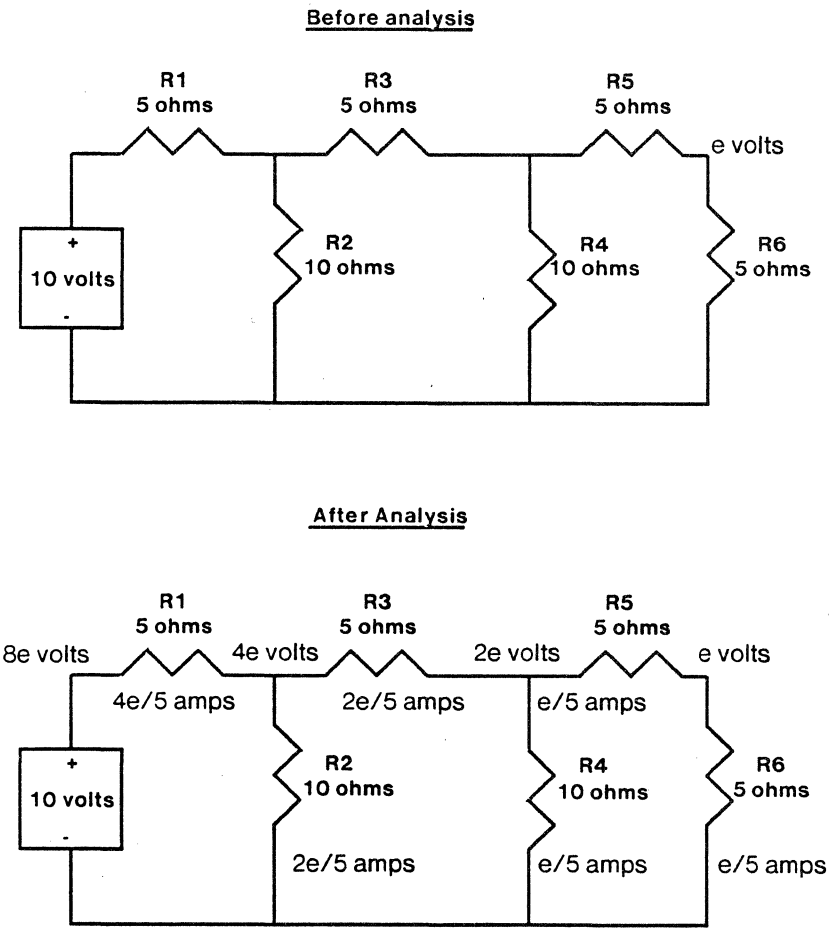


Figure 9. Symbolic propagation of electrical parameters. Analysis begins by assigning the symbol e to the unknown voltage at the uppler right corner of the ladder. Other values are derived by stepwise application of Ohm's and Kirchoff's laws.

Sometimes circuit analysis requires the introduction of more than one variable to represent unknown circuit parameters. In general, the analysis involves two processes: 1-step deductions and coincidence. The 1-step deductions are direct applications (sometimes symbolic) of the electrical laws. A coincidence occurs when a 1-step deduction is made which assigns a value to a circuit parameter that already has a value (symbolic or numeric). At the time of a coincidence, it is often possible to solve the resulting equation for one variable in terms of the others. This allows EL to eliminate unknowns.

The propagation method can be extended to any devices where the electrical laws are invertible and where the algebra required for the symbolic reasoning is tractable. Unfortunately, many simple and important electrical devices, such as inverters and transistors, are too complicated for this approach. For example, a diode is approximately represented by exponential equations. Electrical engineering has an approach for these devices called the *method of assumed states.* This is where guessing enters into EL's problem solving.

The method of assumed states uses a piecewise linear approximation for complicated devices. The method requires making an assumption about which linear region a device is operating in. EL has two possible states for diodes (*on* or *off*) and three states for transistors (*active*, *cutoff*, and *saturated*). Once a state is assumed, EL can use tractable linear expressions for the propagation analysis as before.

After making an assumption, EL must check whether the assumed states are consistent with the voltages and currents predicted for the devices. Incorrect assumptions are detected by means of a contradiction, which is the event in which chosen assumptions are seen to be inconsistent. When this happens, then some of the assumptions need to be changed. Intelligent processing of contradictions involves determining which assumptions to revise. Implementing this idea in the problem-solver led to queue-based control and dependency-directed backtracking as architectural features.

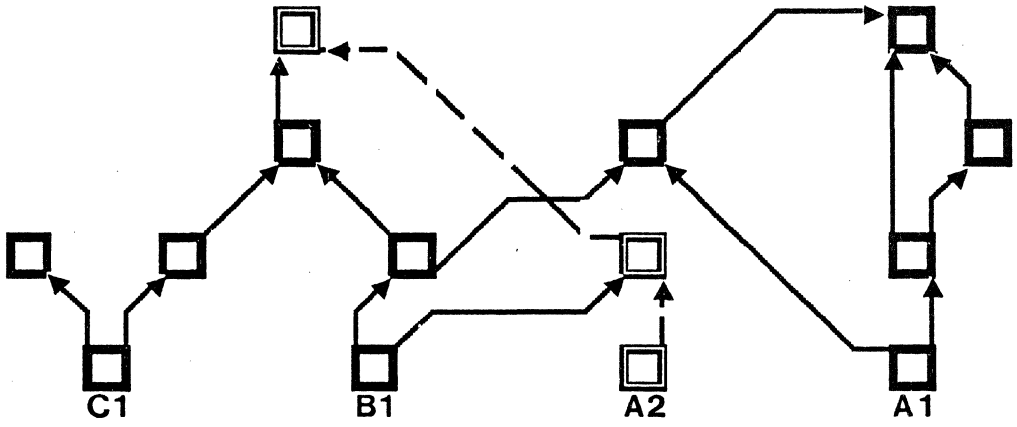
The operators in EL that perform the propagation analysis are called demons. They are placed in queues and run sequentially by a scheduler. When demons run, they make assertions in the data base and then return to the scheduler. This data base activity causes other demons whose triggers match the assertions to be added to the queue.

EL has three queues for DC analysis with different priorities. The lowest priority queue is used for device-state assumptions. These demons are given a low priority so that the immediate consequences of an assumption will be explored before more assumptions are made. The middle queue is used for most of the electrical laws. The high priority queue is used for demons that detect contradictions. These demons are given a high priority so that invalid assumptions will be detected before too much computational work is done. These demons trigger the dependency-directed backtracking.

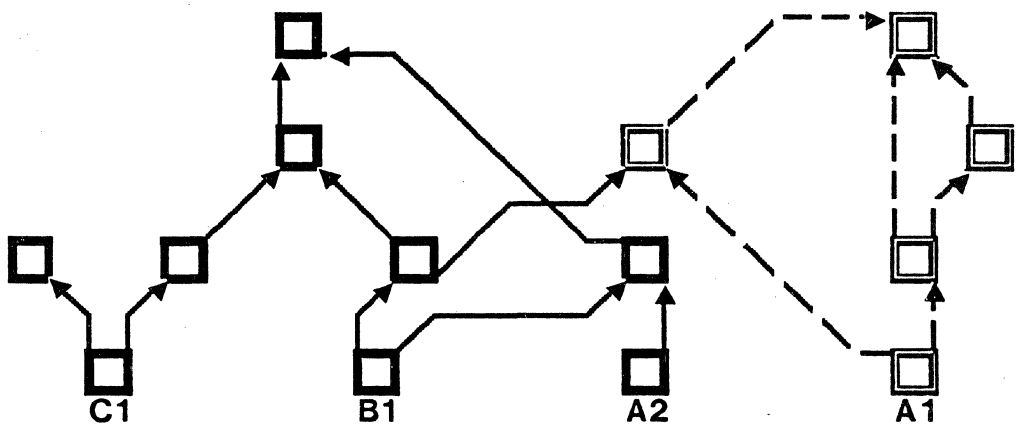
EL keeps dependency records of all of its deductions and assumptions. In EL, an assertion is believed (or *in*) if it has well-founded support from atomic assumptions. An assertion without such support is said to be *out*. If an *out* assertion returns to favor, it is said to be *unouted*. Figure 10 shows an example of this process in a data base. A1, B1, and C1 are atomic data that are currently in. Suppose that A1 and A2 are mutually exclusive device-state assumptions. The top of Figure 10 shows which facts are in when A1 is in. Assertions following from A2 are shown in dotted lines to show that they are in the data base, but that they are out. Arrows indicate support. The bottom figure shows what happens if A1 is outed and A2 is unouted. These ideas are the intellectual precursors to subsequent work on belief revision systems.

An important aspect of EL's problem-solving is its ability to recover from tentative assumptions. The details of the implementation and knowledge will not be covered here, since there are many ways to approach this problem. The main points are:

1. In the event of a contradiction, EL needs to decide what to withdraw. It is not effective to simply withdraw all of the assumptions that are antecedants of the contradictory assertion. EL must decide which of the assumptions are most *unlikely* to change and this requires domain-specific knowledge.



(a) Assumption A1 is in and A2 is out.



(b) Assumption A2 is in and A1 is out.

Figure 10. Example of belief revision in EL. The dark boxes are in, and the lighter ones are out. In (a), A1 is in and so all of its consequences are also in. In (b), A1 is out but A2 is in.

2. EL must redo some of the propagation analysis. Sometimes it is possible to salvage some of the symbolic manipulation (e.g., variable elimination) that has been done. EL has special demons that decide carefully what to forget.
3. Contradictions are remembered so that choice combinations that are found to be inconsistent are not tried again.

In summary, EL is an example of a program with organizational provisions for plausible reasoning. It uses symbolic forward reasoning for analyzing circuits. To analyze complicated devices EL has to assume linear operating regions. It uses dependency-directed backtracking so that it can recover from incorrect assumptions. It uses a priority-oriented queue to schedule tasks so that contradictions will be found quickly and so that the immediate consequences of assumptions will be considered before further assumptions are made.

4.9 Case 9 -- Single Line of Reasoning is Too Weak

When we explain to someone how we solved a problem, we often invoke 20-20 hindsight and leave out the mistakes that we made along the way. Our explanation makes it appear that we followed a very direct and reasonable route from beginning to end. For developing intuitions about problem-solving behavior, this gives a misleading impression that problem-solving is the pursuit of a single line of reasoning. Actually, there are important and somewhat subtle reasons for being able to use multiple lines of reasoning in problem solving and several of the systems described above gain power from this ability. These systems use multiple lines of reasoning to broaden the coverage of an incomplete search or to combine the strengths of separate models.

The HEARSAY-II system [Erman80] provides the best example of the first purpose. (It is described in the next section). In coping with the conflicting demands of searching a large space with limited computational resources, HEARSAY-II performs a heuristic and incomplete search. In general, programs that have fallible evaluators can decrease the chances of discarding a good solution from weak evidence by carrying a limited number of solutions in parallel.

A good example of combining the strengths of multiple models is given by the SYN program reported by Sussman, Steele, and de Kleer [Sussman80], [de Kleer80]. SYN is a program for circuit synthesis, that is, for determining values for components in electrical circuits. The EL program described in the previous section determined circuit parameters such as voltage given fully specified components in a circuit. SYN determines values for the components (e.g., the resistance of resistors) given the form of the circuit and some constraints on its behavior.

SYN uses many of the propagation analysis ideas developed for EL. The interesting new organizational idea in SYN is the idea of *slices* or multiple views of a circuit. This corresponds to the idea of equivalent circuits in electrical engineering practice. A simple example of a slice is the idea that a voltage divider made from two resistors in series can be *viewed* as a single resistor; one slice of the circuit describes it as two resistors and another slice describes it as one. To analyze the voltage divider, SYN uses the second slice to compute the current through the divider. Then by reverting back to the first slice, SYN can compute the voltage at the midpoint. In general, the idea is to switch to equivalent representations of circuits to overcome blockages in the propagation of

constraints. The power of slices is that they provide redundant paths for information to travel in propagation analysis.

By exploiting electrically equivalent forms of circuits involving resistors, capacitors, and inductances, SYN is capable of analyzing rather complex circuits without extensive algebraic manipulation. The idea of slices is not limited to electrical circuits. For example, algebraic transformations of equations can be viewed as means for shifting perspectives. Sussman and Steele also give an example of understanding a mechanical watch by using structural and functional decompositions.

In summary, slices are used to combine the strengths of different models. When they are combined with forward reasoning, they provide redundant paths for information to propagate. A problem-solver based on this idea must know how to create and combine multiple views.

4.10 Case 10 -- Single Source of Knowledge is Too Weak

An important adjunct to the use of multiple reasons in problem solving is the use of multiple sources of knowledge. In this section we will consider the HEARSAY-II system, which coordinates diverse sources of knowledge using an opportunistic scheduler. HEARSAY-II is a system for speech understanding reported by Erman *et al.* [Erman80]. It recognizes spoken requests for information from a data base. Production of speech involves a series of transformations starting with the speaker's intentions, through choice of semantic and syntactic structures, and ending with sound generation. To understand speech HEARSAY-II must reverse this process.

In HEARSAY-II the knowledge for understanding speech is organized as a set of interacting modules (called knowledge sources or KSs) as shown by the arrows in Figure 11. The KSs cooperate in searching a multi-level space of partial solutions. They extract acoustic parameters, classify acoustic segments into phonetic classes, recognize words, parse phrases, and generate and evaluate hypotheses about undetected words and syllables. The KSs communicate through a global data base called a *blackboard* with seven information levels as shown in the figure. These levels are HEARSAY-II's heterogenous abstraction spaces. The primary relationship between levels is compositional: word sequences are composed of words, words are composed of syllables, and so on. Entitities on the blackboard are hypotheses. When KSs are activated, they create and modify hypotheses on the blackboard, record the evidential support between levels, and assign credibility ratings. For example, a sequence of acoustic segments can be evidence for identifying a syllable in a specific interval of the utterance; the identification of a word as an adjective can be evidence that the following word will be an adjective or noun.

HEARSAY-II's use of abstraction differs from the systems considered in cases 5 through 8. Those systems all use uniform abstraction spaces. The abstractions are uniform in that they use the same vocabulary as the final solutions and differ only in the amount of detail. For example in the planning systems, abstract plans have the same structure and vocabulary as final plans. In HEARSAY-II, the diversity of knowledge needed to solve problems justifies the use of heterogeneous abstraction spaces.

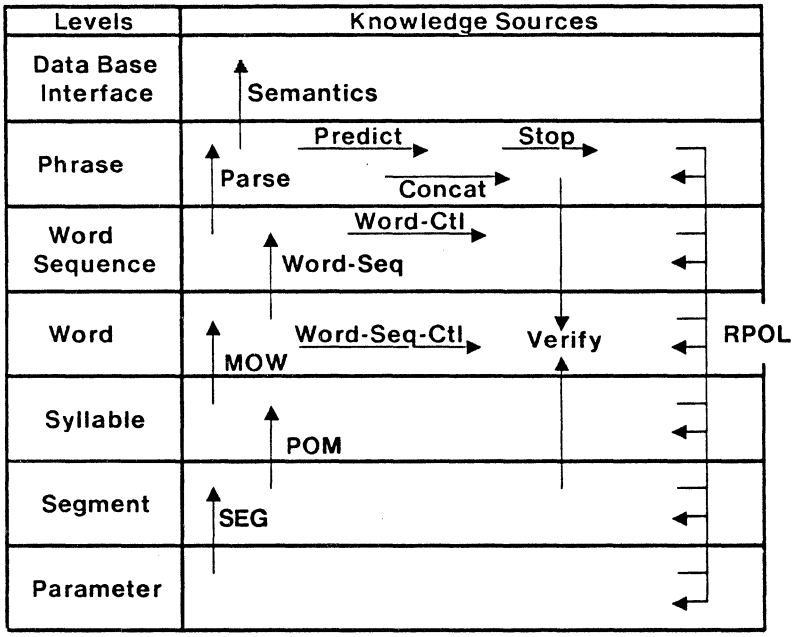


Figure 11. Levels and Knowledge Sources in HEARSAY-II.

The knowledge sources are as follows:

Semantics:	Generates interpretation for the information retrieval system.
SEG:	Digitizes the signal, measures parameters, produces labeled segmentation.
POM:	Creates syllable-class hypotheses from segments.
MOW:	Creates word hypotheses from syllable classes.
Word-Ctl:	Controls the number of hypotheses that MOW makes.
Work-Seq:	Creates word-sequence hypotheses for potential phrases.
Word-Seq-Ctl:	Controls the number of hypotheses that Word-Seq makes.
Predict:	Predicts words that follow phrases.
Verify:	Rates consistency between segment hypotheses and contiguous word-phrase pair.
Concat:	Creates a phrase hypothesis from a verified contiguous word-phrase pair.
RPOL:	Rates the credibility of hypotheses.

A computational system for understanding speech is caught between three conflicting requirements: a large space of possible messages to understand, variability in the signal, and the need to finish in a limited amount of time. The number of possible *ideal messages* is a function of the vocabulary, language constraints, and the semantics of the application. The number of actual messages that a system encounters is much larger than this because speech is affected by many sources of variability and noise. At the semantic level, errors correspond to peculiarities of conceptualization. At the syntactic level errors correspond to peculiarities of grammar. At the lexical and phonemic levels the variance is in word choice and articulation. Errors at the lower levels compound difficulties at the high levels. For example, the inability to distinguish between the four phrases:

till Bob rings
tell Bob rings
till Bob brings
tell Bob brings

may derive from ambiguities in the acoustic levels. HEARSAY-II copes with this by getting the KSs at different levels to cooperate in the solution process. In doing this, HEARSAY-II combines both top-down and bottom-up processing and reasons about resource allocation with a process called opportunistic scheduling.

An example of top-down processing is the reduction of a general sentential concept into alternate sentence forms, each sentence form into specific alternative word sequences, specific words into alternative phone sequences and so on until a best interpretation is identified. Bottom-up processing tries to synthesize interpretations from the data. For example, one might combine temporally adjacent word hypotheses into syntactic or conceptual units. In HEARSAY-II, some KSs use top-down processing and other KSs use bottom-up processing.

All KSs compete to be scheduled and HEARSAY-II tries to choose the most promising KSs at any given moment using opportunistic scheduling. Opportunistic scheduling combines the idea of least commitment with strategies for managing limited computational resources. The opportunistic scheduler adapts automatically to changing conditions of uncertainty by changing the breadth of search. The basic mechanism for this is the interaction between KS-assigned credibility ratings on hypotheses and scheduler-assigned priorities of pending KS activations. When hypotheses have been rated equally, KS activations for their extension are scheduled together. In this way, ambiguity between competing hypotheses causes HEARSAY-II to search with more breadth, and to delay the choice among competing hypotheses until more information is brought to bear.

HEARSAY-II's approach to data interpretation differs from that of GA1 discussed in Section 4.4. Both programs contend with very large search spaces. Both programs need to have effective ways to rule out large classes of solutions. GA1 does this with early pruning. In the absence of constraints, it would expand every solution in the space. HEARSAY-II constructs a complete solution by extending and combining partial candidates. Because of its opportunistic scheduler, it heuristically selects a limited number of partial candidates to pursue. To avoid missing solutions, HEARSAY-II must not focus the search too narrowly on the most "promising" subspaces.

In summary, HEARSAY-II provides an example of an architecture created to meet several conflicting requirements. Multiple levels provide the necessary abstractions for searching a large space. The levels are heterogeneous to match the diversity of the interpretation knowledge. Opportunistic scheduling combines the least commitment idea with the ability to manage computational resources by varying the breadth of search and by combining top-down and bottom-up processing.

4.11 Case 11: General Representation Methods are Too Inefficient

Research on expert systems has benefited from the simplicity of using uniform representation systems. However, as knowledge bases get larger, the efficiency penalty incurred by using declarative and uniform representations can become significant. Attention to these matters will become increasingly important in ambitiously conceived future expert systems with increasingly large knowledge bases.

This section considers architectural approaches for tuning the performance of expert systems by making changes to the representation of knowledge. This section considers specialized data structures, knowledge compilation and knowledge transformations for cognitive economy.

In Section 3, we showed that symbols can have interpretations and that computers can perform inferences by manipulating symbol structures. We suggested that symbol structures can be represented as list structures, but we did not consider that the organization of list structures as data structures has consequences for the efficiency of retrieving information. The selection and creation of efficient data structures is a principal part of most computer science curriculums and many textbooks are available [Knuth]. Consequently, several of the programs discussed in the previous cases (e.g., DENDRAL, GA1, and HEARSAY-II) use specialized data structures. In general, these data structures are designed so that facts that are used together are stored together and facts that are used frequently can be retrieved efficiently.

Choice of data structure depends on assumptions about how the data will be used. A common assumption about special data structures is that they are complete with regard to specific relationships. For example, in GA1's data structure for molecular hypotheses, molecular segments are connected if and only if they are linked in the data structure. This sort of assumption is commonplace in representations like maps, which are assumed to show *all* of the streets and street intersections. Representations whose structure in the medium is analogous to the structure of the world being modeled are sometimes called analogical representations.

A less understood issue is the use and selection of data structures in systems where many kinds of information are used. There is not much experience with systems that mix a variety of different representations. One step in this direction is to tag relations with information describing the chosen representation so that specialized information can be accessed and manipulated using uniform mechanisms. Some ideas along this line appeared in Davis' thesis [Davis76a], where schemata were used to describe some formatting and computational choices, but the work has not been extended to describe general dimensions of representation [Bobrow75] for use by a problem solver.

Another important idea for knowledge bases is the idea of *compiling* knowledge. By compilation, we mean any process which transforms one representation of knowledge into another representation that can be used more efficiently. This transformation process can include optimization as well as the tailoring of representations for particular instruction sets. Space does not permit a detailed discussion of techniques here, but some examples are listed below to suggest the breadth of the idea.

Example 1. Burton reported a system for taking ATN grammars and compiling them into executable code [Burton76]. The compiled grammars could be executed to parse sentences much more rapidly than previous interpreter-based approaches.

Example 2. Production system languages have been studied and experimented with for several years ([Davis76b]. A basic difficulty with many production systems is that large production system programs execute more slowly than small ones. The extra instructions do not need to execute to slow down the system; their mere presence interferes with the matching process that selects productions to run. An example of one such language is OPS2 reported by Forgy and McDermott [Forgy77]. Forgy conducted a study of ways to make such production systems more efficient by compiling them into a network of "node programs" [Forgy79]. The compiler exploits two properties of production systems: structural similarity and temporal redundancy. Structural similarity refers to the fact that many productions contain similar conditions; temporal redundancy refers to the fact that individual productions change only a few facts in the memory so that most of the data is unchanged from cycle to cycle. Forgy's RETE matching process exploits this by looking only at the *changes* in the memory. Forgy's analysis shows how several orders of magnitude of speed up can be achieved by compiling the productions and by making some simple changes to computing hardware.

Example 3. Another system that compiles a knowledge base of production rules, EMYCIN, was reported by van Melle [van Melle80]. EMYCIN is not considered a "pure" production system since it is not strictly data-driven; the order that the rules are tried in EMYCIN is controlled by the indexing of parameters. This means that EMYCIN's interpreter does not repeatedly check elements in the working memory so that some of the optimizations used by Forgy would provide much less of an improvement for EMYCIN. EMYCIN's rule compiler concentrates on eliminating redundancy in the testing of similar patterns in rules and compiles them into decision trees represented as LISP code.

Example 4. The HARPY system for speech recognition reported by Lowerre [Lowerre76] illustrates several issues about compilation. HARPY represents the knowledge for recognizing speech in a unified data structure (context-free production rules) which represents the set of all possible utterances in HARPY's domain. This data structure represents essentially the same information that was used in HEARSAY-II except for the parameterization and segmentation information. HARPY's knowledge compiler combines the syntax, lexical, and juncture knowledge into a single large transition network. First, it compiles the grammar into a word network, then it replaces each word with a copy of its pronunciation graph and inserts word-juncture rules at the word boundaries. In the final network, each path from a start node to an end node represents a sequence of segments for some sentence. With the knowledge in its compiled form, HARPY is capable of performing a

rapid search process that attempts to find the best match between the utterance and the set of interpretations. The major concerns about the extensibility of this idea are (1) that the highly stylized form of the input that HARPY can accept makes it difficult to add new knowledge and (2) the compilation is expensive for a large knowledge base (13 hours of PDP-10 time for a thousand word, grammar).

The promise of knowledge compilation ideas is to make it possible to use very general means for representing knowledge while an expert system is being built and debugged. Then a compiler can be applied to make the knowledge base efficient enough to compete with hand coding. Given a compiler, there is no need to sacrifice flexibility for efficiency: the knowledge base can be changed at any time and recompiled as needed. In addition, the compiler can be modified to re-represent the knowledge efficiently as hardware is changed, or as the trade-offs of representation become better understood. The techniques for doing this are just beginning to be explored and will become increasingly important in the next few years.

So far in our discussion of efficiency we have assumed that it is necessary for the designers of a knowledge base to anticipate how knowledge will be used and to arrange for it to be represented efficiently. Lenat, Hayes-Roth, and Klahr [Lenat79] coined the term cognitive economy to refer to systems which automatically improve their performance by changing representations, changing access (e.g., caching), and compiling knowledge bases. Systems like this need to be able to predict how representations should be changed, perhaps by measurements on representative problems. The ideas of cognitive economy and knowledge compilation are more speculative than the other ideas we have considered and there are many theoretical and pragmatic issues to be resolved before they will be practical.

5. SUMMARY OF KNOWLEDGE ENGINEERING CASES

Our pedagogical tour of cases began with the consideration of a very simple architecture. It was suitable for problems having a small solution space and reliable, constant data and knowledge. Few real problems satisfy all of these requirements.

In the second case we considered ways to cope with unreliable data or knowledge. Probabilistic, fuzzy, and exact methods were discussed. All of these methods are based on the idea of increasing reliability by combining evidence. Each method requires the use of meta-knowledge about how to combine evidence. The probabilistic (and pseudo-probabilistic) approaches use various *a priori* and conditional probability estimates; the fuzzy approaches use fuzzy set descriptions; the exact approaches use non-monotonic data correction rules. Errorful data and knowledge seem to be ubiquitous in real applications. In case ten, we returned to this issue to discuss how an opportunistic scheduler can be used to cope with the conflicting requirements of errorful data, limited computational resources, and a large search space by varying the breadth of a heuristic search.

In the third case we considered time-varying data. We started with the situational calculus and then discussed a program that used transition rules to trigger expectations in a monitoring task. More sophisticated ways to reason with time seem to require more research.

The remaining cases dealt with ways to cope with large search spaces. We started with the hierarchical generate and test approach in the fourth case. This requires a solution space to be *factorable* in order to allow early pruning. This approach explores the space of solutions systematically and can be quite effective for returning all consistent solutions.

Generate and test is a weak method, applicable only when early pruning is feasible. It requires the ability to generate candidates in a way that allows large classes to be pruned from very sparse partial solutions. In many applications, solutions must be instantiated in substantial detail before they can be ruled out. Fortunately, it is not necessary in all applications to consider all possible solutions and to choose the best; satisficing is sufficient. The next few cases describe reasoning with abstractions to reduce the combinatorics without early pruning. The methods presented are usually applicable in satisficing tasks.

In case five, we considered a form of reasoning based on fixed abstractions. This approach requires that all of the information needed for testing partial solutions be available before a subproblem is generated. This requirement exposes the weakness of this approach: some problems can not be solved from the available information without backtracking. While the abstractions make the problem solver efficient, their use is too rigid for some applications.

Top-down refinement is more flexible. Abstractions are composed from a set of concepts in a hierarchical space. The simplest version of top-down refinement (case six) uses a fixed criticality ordering on the concepts and a fixed partial order for solving subproblems.

Top-down refinement does not allow for variability in the readiness to make decisions. In the seventh case, we introduced the least commitment principle which says that decisions should be

postponed until there is enough information. This approach tends to exploit the synergy of interactions between subproblems. It requires the ability to suspend activity in subproblems, move information between subproblems, and then restart them as information becomes available. In this case, the problem-solving knowledge is much richer than in the previous methods. It was suggested that principles like least commitment should be incorporated as part of meta-level problem solving.

An inherent difficulty with pure least commitment approaches is the phenomenon of a least commitment deadlock. When a problem solver runs out of decisions that it can make immediately, it must make a guess to continue. The amount of guessing is a measure of its missing knowledge, and knowledge bases are usually incomplete. This theme was continued in case eight where dependency-directed backtracking was discussed as an approach supporting efficient retraction of beliefs in plausible reasoning.

Cases nine and ten illustrated the use of multiple lines of reasoning to enhance the power of a problem solver and the use of heterogenous abstraction models to capture the variety of knowledge in some applications. An opportunistic scheduler uses knowledge sources as soon as they become applicable (either top-down or bottom-up) and controls the breadth of search.

Finally, we considered some methods for speeding up processing and information retrieval: specialized data structures and knowledge compilers. These techniques do not attack the basic combinatorics of search, but they do reduce the *cycle time* of problem solvers and will become increasingly important in future expert systems with large knowledge bases.

In articulating these ideas about expert systems, we are forced to decide what is essential and important about previous systems. In some cases, other expert systems could have been substituted to make the same points. Our view is that the recent critical work in expert systems has focused on the mechanisms of problem-solving, and research into this area has been the most fruitful when it has been directed towards substantial applications. In these applications, both knowledge bases and methods were tested and revised.

A system is always a product of its time. System builders operate against a background of competing ideas and controversies and must also confront the limitations of their resources. To summarize experiments and create a simplified theory, we must necessarily step outside of this rich historical context. Inescapably, we are bound to our current vantage point. As this paper is written, there is a ferment of activity in expert system research. The theory of building intelligent systems is far from complete and the ideas expressed here are by no means universally accepted in the AI community. To wait for the ideas to settle and survive the test of history would preclude creating a timely guide to current thinking.

REFERENCES

[Barr80]
Barr, A. & Feigenbaum, E. A. *The handbook of artificial intelligence*, Computer Science Department, Stanford University, 1980.

[Buchanan78]
Buchanan, B. G., & Feigenbaum, E. A., DENDRAL and Meta-DENDRAL: Their applications dimension, *Artificial Intelligence* **11**, pp. 5-24, 1978.

[Bobrow77]
Bobrow, D. G., Winograd, T. An overview of KRL, a knowledge representation language, *Cognitive Science*, **1:1**, January 1977, pp. 3-46.

[Bobrow75]
Bobrow, D. G. Dimensions of representation. in Bobrow, D. G. & Collins, A. (Eds.) *Representation and Understanding*, New York: Academic Press, 1975.

[Burton76]
Burton, R. R. *Semantic grammar: an engineering technique for constructing natural language understanding systems*, Bolt Beranek and Newman Report No. 3453, December 1976.

[Charniak80]
Charniak, E., Riesbeck C. K., & McDermott, D.V. *Artificial intelligence programming*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1980.

[Davis77]
Davis, R., Buchanan, B. G., & Shortliffe, E. Production rules as a representation for a knowledge-based consultation program. *Artificial Intelligence*, **8**, 1977, pp. 15-45.

[Davis76a]
Davis, R. Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases, Ph.D. thesis, Stanford University, AI LAb Memo AIM-283 (July 1976).

[Davis76b]
Davis, R. & King, J. An overview of production systems. in E. W. Elcock and D. Michie (Eds.), *Machine Intelligence*, New York: Wiley, 1976, pp. 300-332.

[de Kleer80]
de Kleer, J., & Doyle J. Dependencies and assumptions, (to appear in [Barr80].)

[de Kleer77]
de Kleer, J., Doyle J., Steele, G. L., Sussman, G. J. Explicit control of reasoning, MIT Memo No. 427, June 1977.

[de Kleer80]
de Kleer, J. & Sussman, G. J. Propagation of constraints applied to circuit synthesis, *Circuit Theory and Applications* **8**, 1980, pp. 127-144.

[Doyle80]
Doyle J. & London P. A selected descriptor-indexed bibliography to the literature on belief revision, *Sigart Newsletter*, **71**, 1980, pp. 7-22.

[Doyle80a]
Doyle, J. A model for deliberation, action, and introspection. AI TR 581, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1980.

[Doyle79]
Doyle, J. A truth maintenance system. *Artificial intelligence* **12**, 1979, pp. 231-272.

[Duda76]
Duda, R. O., Hart, P. E., Nilsson, N. J. *Subjective bayesian methods for rule-based inference systems*, SRI International, Artificial Intelligence Center Technical Note 124, January 1976.

[Erman80]
Erman L. D., Hayes-Roth, F., Lesser, V. R., Reddy, D. R. The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. *ACM Computing Surveys.* **12:2**, June 1980.

[Fagan79]
Fagan, L. M., Kunz, J. C., Feigenbaum, E. A., Osborn, J. J. Representation of dynamic clinical knowledge: measurement interpretation in the intensive care unit, *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, August, 1979.

[Fagan80]
Fagan, L. M., *VM: Representing time-dependent relations in a medical setting*, doctoral dissertation, Stanford University, Computer Science Department, June 1980.

[Feigenbaum77]
Feigenbaum, E. A. The art of artificial intelligence: I. Themes and case studies of knowledge engineering, *Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 1014-1029.

[Fikes70]
Fikes, R. E. REF-ARF: A system for solving problems stated as procedures, *Artificial Intelligence*, **1**, 1970, pp. 27-120.

[Forgy77]
Forgy, C. & McDermott, J. OPS: A domain-independent production system, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 933-939.

[Forgy79]
Forgy, C. A. *On the efficient implementation of production systems*, doctoral dissertation, Carnegie-Mellon University, Department of Computer Science, February 1979.

[Gardner79]
Gardner, A. Search. in [Barr80], (Also Stanford University Computer Science Department Report No. STAN-CS-79-742, June 1979).

[Hayes-Roth79]
Hayes-Roth, B. & Hayes-Roth, F. A cognitive model of planning, *Cognitive Science*, 1979, 3, pp. 275-310.

[Kleene67]
Kleene, S. C. *Mathematical Logic.* New York: John Wiley & Sons, 1967.

[Knuth]
Knuth, D. *The art of computer programming*, Addison-Wesley Publishing Company.

[Lenat79]
Lenat, D. B., Hayes-Roth, F., Klahr, P. *Cognitive Economy*, Stanford University, Computer Science Deparment, Heuristic Programming Project Report HPP-79-15, June 1979.

[Lowerre76]
Lowerre, B. T., *The HARPY Speech Recognition System*, Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1976.

[London78]
London, P. E., Dependency networks as a representation for modelling in general problem solvers, University of Maryland Technical Report TR-698 NSG-7253. September 1978.

[McAllester80]
McAllester, D. A. An outlook on truth maintenance, MIT AI Memo No. 551, April 1980.

[McCarthy69]
McCarthy, J. & Hayes, P. J., Some philosophical problems from the standpoint of artificial intelligence, in Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence* 4, 1969, Halsted Press, pp. 463-502.

[McDermott80]
McDermott, J. R1: A rule-based configurer of computer systems. Carnegie-Mellon University, Department of Computer Science Report CMU-CS-80-119, April 1980.

[Minsky67]
Minsky, M. L., *Computation: Finite and infinite machines.*, Englewood Cliffs, N. J.: Prentice-Hall Inc., 1967.

[Minsky61]
Minsky, M. Steps toward artificial intelligence. in E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*, New York: McGraw-Hill, 1961.

[Newell75]
Newell, A. & Simon, H. A. Computer science as empirical enquiry: symbols and search, *Communications of the ACM*, **19:3**, March 1976.

[Newell73]
Newell, A. Artificial intelligence and the concept of mind. in R. C. Schank and K. M. Colby (Eds.), *Computer Models of Thought and Language.* San Francisco: W. H. Freeman, 1973.

[Newell62]
Newell, A. Some problems of basic organization in problem-solving programs. In M. C. Yovits, G. T. Jacobi, G. D. Goldstein (eds.) *Proceedings of the Second Conference on Self-Organizing Systems*, Washington D.C.: Spartan Books, 1962, pp. 393-423.

[Nilsson80]
Nilsson, N. J. *Principles of artificial intelligence.* Palo Alto, California: Tioga Publishing Company, 1980.

[Nilsson71]
Nilsson, N. J. *Problem-solving methods in artificial intelligence.* New York: McGraw-Hill Book Company, 1971.

[Pednault81]
Pednault, E. P. D., Zucker, S. W., & Muresan, L. V. On the independence assumption underlying subjective Bayesian updating, *Artificial Intelligence* **16**, 1981, pp. 213-222.

[Sacerdoti77]
Sacerdoti, E. D. *A structure for plans and behavior.* New York: American Elsevier, 1977. (originally published 1975).

[Sacerdoti74]
Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 1974, **5:2**, pp. 115-135.

[Shortliffe78]
Shortliffe, E. H., Buchanan, B. G., Feigenbaum, E. A., Knowledge engineering for medical decision making: A review of computer-based clinical decision aids, *Proceedings of the IEEE*, **67**, September 1974, pp. 1207-1224.

[Shortliffe76]
Shortliffe, E. H. *Computer-based medical consultations: MYCIN.* New York: American Elsevier, 1976.

[Stallman77]
Stallman, R. M. & Sussman, G. J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* **9**, 1977, pp. 135-196.

[Stefik82]
Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E. The organization of expert systems: a tutorial, *Artificial Intelligence* (in press), 1982.

[Stefik81a]
Stefik, M. J. Planning with constraints. *Artificial Intelligence* **16:2**, May 1981, pp. 111-140.

[Stefik81b]
Stefik, M. J. Planning and meta-planning. *Artificial Intelligence* **16:2**, May 1981, pp. 141-170. (Also preprinted in *Readings in AI* by Webber and Nilsson.)

[Stefik78]
Stefik, M. Inferring DNA structures from segmentation data. *Artificial Intelligence* **11**, pp. 85-114, 1978.

[Suppes57]
Suppes, P. *Introduction to Logic.* Princeton: D. Van Nostrand Company, 1957.

[Sussman80]
Sussman, G. J. & Steele, G. L. CONSTRAINTS -- A language for expressing almost-hierarchical descriptions. *Artificial Intelligence* **14**, 1980, pp. 1-39.

[van Melle80]
van Melle, W. *A domain-independent system that aids in constructing knowledge-based consultation programs*, doctoral dissertation, Stanford University, Computer Science Department Report No. STAN-CS-80-820, June 1980.

[Weyhrauch80]
Weyhrauch, R. W. Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence*, **13:1-2**, April 1980.

[Wilensky80]
Wilensky, R. Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science* 5:3, July-Sept 1981, pp. 197-233.

[Winograd80]
Winograd, T. Extended inference modes in reasoning by computer systems. *Artificial Intelligence*, **13:1-2**, April 1980.

[Zadeh79a]
Zadeh, L. A. A theory of approximate reasoning. in Hayes, J. E., Michie, D., and Mikulich, L. I. (Eds.), *Machine Intelligence 9*, New York: John Wiley & Sons, Inc., 1979

[Zadeh79b]
Zadeh, L. A., Possibility theory and soft data analysis, University of California at Berkeley, Electronics Research Laboratory, Memorandum No. UCB/ERL M79/66, August 27, 1979.